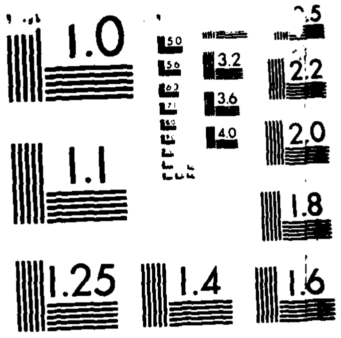
AD-A164 043 CONTROL CIRCUITRY FOR HIGH SPEED VLSI (VERY LARGE SCALE 1/4
INTEGRATION) WINO. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI. P C ROSSBACH
UNCLASSIFIED DEC 85 AFIT/GE/ENG/85D-35 F/G 9/5 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
3.2
3.6
4.0

1

AF INSTITUTE OF TECHNOLOGY

CONTROL CIRCUITRY FOR HIGH SPEED VLSI WINOGRAD FOURIER TRANSFORM PROCESSORS

THESIS

Paul C. Rossbach
Captain, US ARMY

AFIT/GE/ENG/85D-35

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GE/ENG/85D-35

CONTROL CIRCUITRY FOR HIGH SPEED VLSI WINOGRAD FOURIER TRANSFORM PROCESSORS

THESIS

Paul C. Rossbach
Captain, US ARMY

AFIT/GE/ENG/85D-35

AFIT/GE/ENG/85D-35

CONTROL CIRCUITRY FOR HIGH SPEED VLSI WINOGRAD FOURIER TRANSFORM PROCESSORS

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirement for the Degree of

Master of Science in Electrical Engineering

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

Paul C. Rossbach, B.S.
Captain, US ARMY

December 1985

Acknowledgments

I would like to acknowledge my contemporaries in the Winograd Fourier Transform Algorithm Processor Design Group whose long, hard efforts have resulted in an exciting, state-of-the-art digital signal processor design. My contemporaries, Capt James Collins, Capt Paul Coutee, Capt Kent Taylor, and I are sure that future efforts on this project will produce the Fourier transform processor system that we have begun to implement.

I would also like to acknowledge and thank my thesis advisor, Capt Richard Linderman. His insight and dedication has made this entire design effort possible, and has allowed and encouraged the design group to learn and accomplish more than any of us thought possible in the short time available.

Table of Contents

List of Figures

List of Tables

AFIT/GE/ENG/85D-35

Abstract

The calculation of the Discrete Fourier Transform (DFT) has long been a significant bottleneck in many Digital Signal Processing applications. With the arrival of Very Large Scale Integration (VLSI) and new DFT algorithms, system architectures that significantly reduce the DFT bottleneck are possible. This study addresses the design, simulation, implementation, and testing of the control circuitry for a high speed, VLSI Winograd Fourier Transform (WFT) processor. Three WFT processors are combined into a pipelined architecture that is capable of computing a 4080-point DFT on complex input data approximately every 120 microseconds when operating with 70 MHz clock signals. The chip control architecture features a special Programmable Logic Array (PLA) to control the on-chip arithmetic circuitry, and a dense, 54K ROM to generate data addresses for the external RAM. The PLA controller was fabricated in 3 micron CMOS and functioned properly for clock rates of over 60 MHz. The address generator ROM was designed and submitted for fabrication in 3 micron CMOS, and SPICE simulations predict an access time of 60 nanoseconds.

Software that automatically generates a ROM layout description from a data file was developed to ensure the correctness of the final design. Software was also developed to optimize the ROM by attempting to minimize the number of transistors required to represent the information. The software further optimizes the ROM by removing any unnecessary drain/source areas. The transistor minimization procedure is based on a graph partitioning heuristic, and the drain removal procedure is based on an algorithm that near-optimally solves the Traveling Salesman Problem. The ROM optimization produces large gains in power, yield, and in some cases speed.

CONTROL CIRCUITRY FOR HIGH SPEED VLSI WINOGRAD FOURIER TRANSFORM PROCESSORS

I. Introduction

Background

The ability to perform detailed signal spectral analysis at an ever increasing rate has been a major goal in the area of digital signal processing since an efficient algorithm for computing Discrete Fourier Transforms (DFTs) was disclosed in 1965 (Cooley and Tukey, 1965). Since the DFT is the central computation in most spectrum analysis problems, fast and efficient methods for its implementation are crucial. Digital systems that can perform rapid spectrum analysis have a number of current applications including speech processing, seismic processing, pattern recognition, artificial intelligence, sonar, radar, and military target acquisition systems.

With the arrival of large scale integration and the resulting reduction in cost and size of digital components, together with increased speed, the performance of digital systems in important application areas is continuing to improve. Additionally, new DFT algorithms and system architectures are constantly being developed to provide even faster and more efficient digital systems. However, the most advanced systems available today are still not able to satisfy all of the military requirements for compact, high speed digital signal processing systems that utilize the DFT.

Therefore, the Air Force Wright Aeronautical Laboratories, Air Force Office of Scientific Research, and Air Force Space Division are sponsoring research for the development of a high speed DFT processor. This DFT processing system must be capable of rapidly computing DFT sizes of up to 4080-points at speeds an order of magnitude faster than systems currently available. Additionally, the processor must be implemented in integrated circuit technology to meet the small size and weight constraints imposed on embedded systems. The desired DFT processing system can be used in numerous applications, but is of prime importance to Synthetic Aperature Radar (SAR) systems. To achieve a more advanced DFT processing system which meets the Air Force requirements, new transform algorithms and hardware architectures must be used while simultaneously increasing data processing throughput.

Problem

The problem addressed in this thesis is how to design, simulate, and implement the control circuitry for an integrated signal processing system that calculates Discrete Fourier Transforms (DFTs) of sizes up to 4080-points at clock rates of 70 MHz. The processing system will make use of the Prime Factor Algorithm (PFA), the Winograd Fourier Transform Algorithm (WFTA), a state-of-the-art 1.2 micron Complementary Metal-Oxide-Semiconductor (CMOS) integrated circuit technology, and special purpose architectural design.

Scope

The thesis design, simulation, and implementation is limited to the control circuitry for the 16-point WFTA processor (WFTA16). The implementation of the control circuitry for the WFTA16 considers the control interfaces required between it, the 15-point, 17-point, and host processor, and facilitates the eventual design and implementation of the control for the 15 and 17-point WFTA chips. WFTA16 control circuit test chips were designed, fabricated, and tested. The WFTA16 control and arithmetic circuits were integrated into a total processor design.

Three concurrent theses address other aspects of the total design project. Taylor presents the PFA and WFTA theory, overall signal processing system architecture, and numerical precision simulation results (Taylor, 1985). Coutee presents the arithmetic circuitry for the WFTA16 (parts of which can be used for the 15 and 17-point WFTAs as well) (Coutee, 1985). Collins presents a validation program for the WFTA16 operation, and describes the WFTA16 modules in VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, VHDL (Collins, 1985).

The contents of the other three theses are crucial to a full and complete understanding of this thesis. Therefore, the four theses should be read in the following order:

1. Taylor
2. Coutee
3. Rossbach
4. Collins

Summary of Current Knowledge

The Prime Factor Algorithm and Winogard Fourier Transform Algorithm. Since the classic paper of Cooley and Tukey (Cooley and Tukey, 1965) appeared, the Fast Fourier Transform (FFT) has been used extensively in many DSP applications. The main advantage of the FFT over the DFT is the substantial reduction in the number of arithmetic operations required. The number of arithmetic operations grows as nlogn for the FFT compared to n^2 for the DFT. In 1978, Winograd presented an algorithm that often uses less than half the number of multiplications required by the FFT and almost the same number of additions (Winograd, 1978). Moreover, the number of multiplications required for the Winograd Fourier Transform Algorithm (WFTA) has been shown to be minimal (Winograd, 1978; Blanken and Rustan, 1982). However, large WFTAs lack the modularity which results in an effective VLSI implementation. A solution to this problem can be derived from a program presented by Burrus that combines the Prime Factor

Algorithm (PFA) of Good and Thomas (Good, 1971) with small Winograd transforms (Burrus and Eschenbacher, 1981). Finally, Linderman presented chip and system architectures that efficiently embed the PFA into VLSI through the use of WFTA processors (Linderman, 1984: Chapter 7).

Overview of the PFA. The PFA changes a one-dimensional problem into an N-dimensional DFT using the prime factor index map of Good and Thomas (Good, 1971; McClellan and Rader, 1979). This approach produces more manageable transform lengths. The Chinese Remainder Theorem (CRT) is used to uniquely map the inputs into an N-dimensional hyper-cube and to map the transform outputs back into the desired one-dimensional result. Since the Chinese Remainder Theorem is used, the factors used to decompose the DFT must be relatively prime (this leads to unusual DFT sizes). Figure I-1 illustrates the mapping where a 6 point one-dimensional transform is computed using a two-dimensional 2 x 3 point transform. The two-dimensional transform consists of three 2-point DFTs followed by two 3-point DFTs. The multidimensional DFT is computed using processors for small, relatively prime DFT sizes. These processors can be implemented using any available algorithm, including the WFTA (Nussbaumer, 1981).

Overview of the WFTA. Winograd's short algorithm is based on cyclic convolution to compute DFTs (Rader, 1968). However, the cyclic convolution is replaced by a

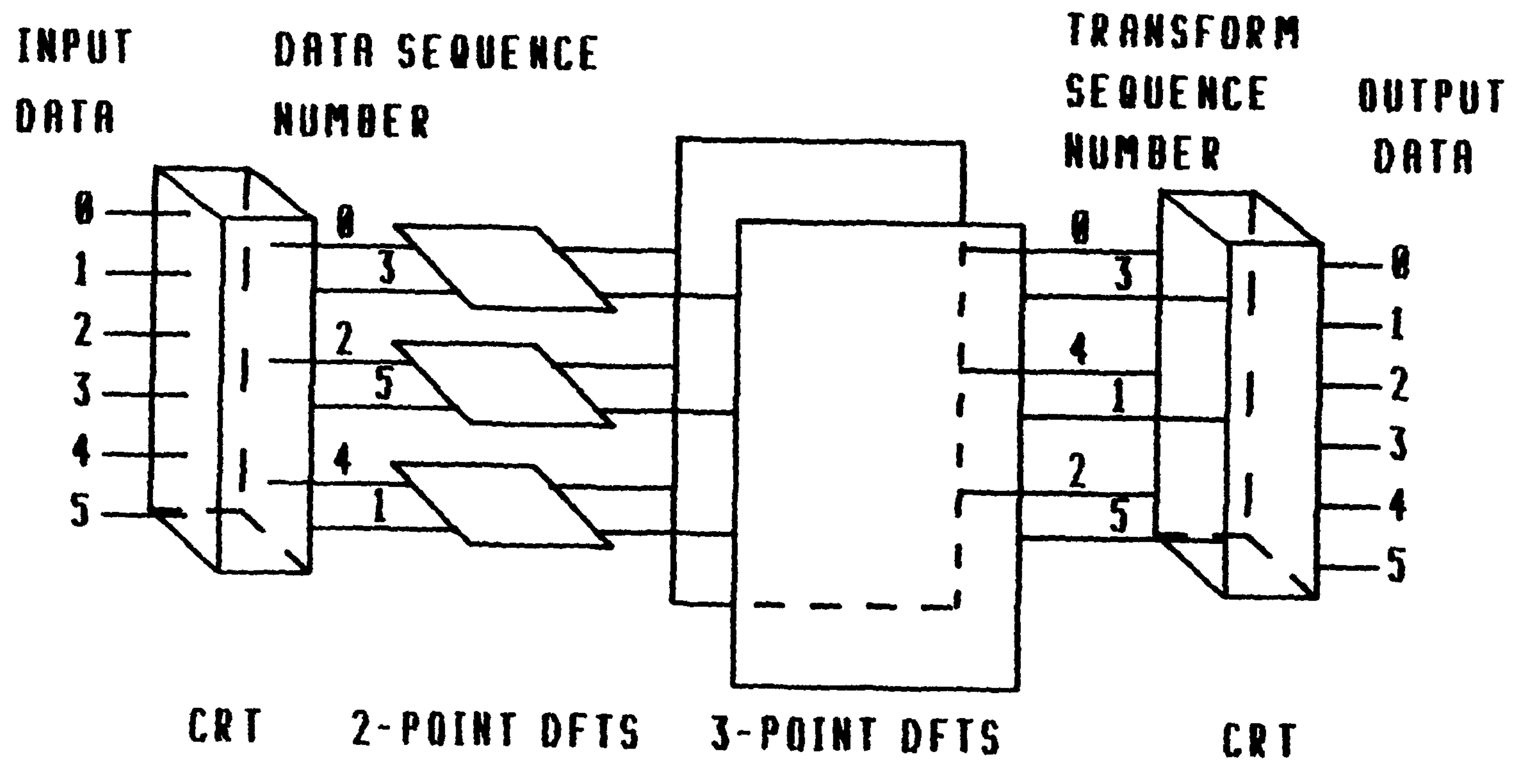


Figure I-1 2 X 3 Two-dimensional Transform series of shorter computations using relatively prime polynomial factors (Blahut, 1985:76-84). Winograd's large algorithm uses smaller WFTA blocks (3, 5) to compute larger DFTs (15). The large algorithm uses associative properties of the Kronecker products of matrices.

The WFTA can be viewed as a sequence of pre-additions, multiplications, and finally post-additions on an input vector to produce the DFT result. This can be written as

$$\bar{X} = C\ D\ A\ \bar{x}$$

where D is a diagonal matrix of fixed coefficients to be multiplied, A and C are rectangular matrices representing the pre-additions and post-additions respectively.

The WFTA is attractive for high throughput VLSI for a number of reasons. First, the number of multiplications is minimal. Second, the matrix representation of the WFTA maps well into VLSI structures for small block sizes (less than 20). Finally, WFTA uses fixed, hard-wired coefficients which results in fewer I/O transfers.

<u>PFA Elements: 15, 16, and 17-Point WFTA Chips</u>. A PFA pipelined architecture that uses 15, 16, and 17-point WFTA processors as the building blocks was presented by Linderman (Linderman, 1981:187-188). Figure I-2 depicts the PFA pipelined architecture that is capable of performing 4080-point DFTs.

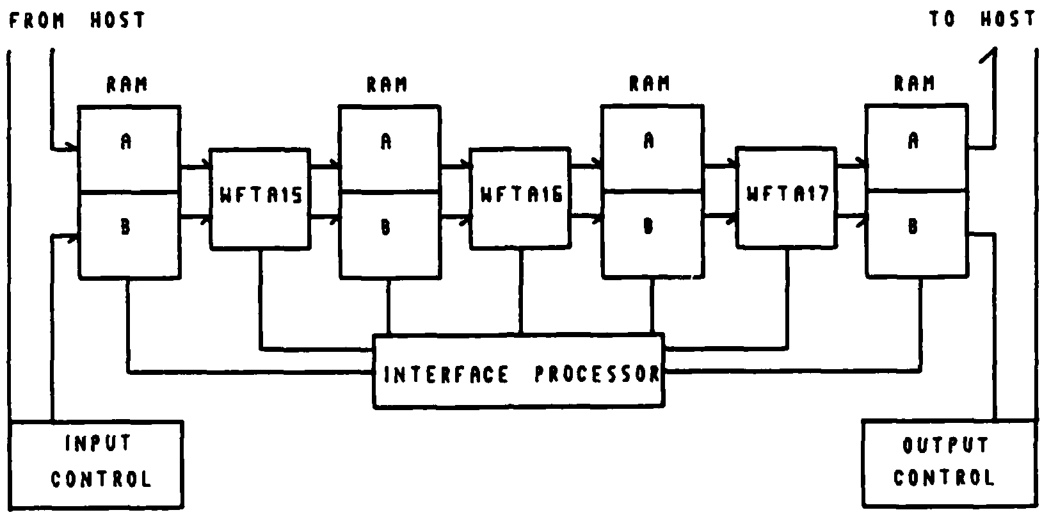


Figure I-2 PFA Pipelined Architecture.

The WFTA processing elements of the PFA pipeline make use of a bit-serial architecture to achieve good numerical performance and high throughput. Additional throughput is obtained by incorporating many serial multipliers that operate in parallel. One key to this increased throughput is the efficiency of the double-bit-serial pipelined multipliers that takes advantage of the constant coefficients in the WFTA.

These special pipelined multipliers make use of a modified Booth's Quaternary Algorithm and are discussed by Coutee (Coutee, 1985).

The PFA pipelined architecture and bit-serial WFTA processing elements proposed by Linderman form the basis and starting point for this thesis (as well as those of Taylor, Collins, and Coutee). The objective is the actual design, implementation, and validation of the WFTA processors and the PFA pipeline architecture.

CMOS Technology. Because of its high switching speeds, low power consumption, and ability to scale to small feature sizes, CMOS is a leading contender for existing and future VLSI systems (Cohen, 1984; Weste and Eshraghian, 1985). The WFTA processing chips will be designed and implemented in the CMOS technology because of its numerous advantages over the NMOS and PMOS technologies. Therefore, a brief discussion of the CMOS technology used throughout the design of the WFTA chips will be presented in the following sections.

CMOS Circuit. CMOS technology provides two types of MOS (Metal-Oxide-Semiconductor) transistors (devices), an n-type transistor (NMOS) and a p-type transistor (PMOS). Both are fabricated in silicon using either negatively doped silicon that is rich in electrons (negatively charged) or positively doped silicon that is rich in holes (positively charged). The physical

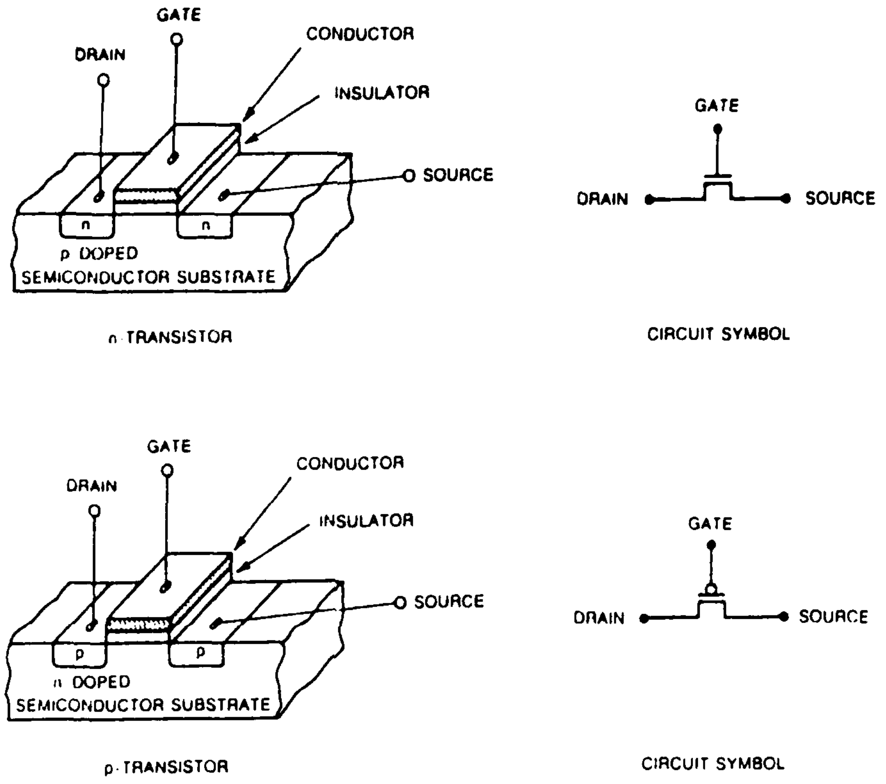


Figure I-3 MOS Transistors (Weste and Eshraghian, 1985:6).

structures for both types of MOS transistors are shown in Figure I-3. The gate of each transistor is a control input, and it controls the flow of current between the source and the drain. The operation of both transistors can be described as that of an on/off switch. The NMOS device is turned on when a "1" (5 volts) is applied to the gate, and turned off when a "0" (0 volts) is applied. The PMOS device is turned on when a "0" is applied to the gate, and off when a "1" is applied. The NMOS and PMOS devices do not always act as perfect or ideal switches. The NMOS device passes a weak "1" and the PMOS device passes a weak "0".

The NMOS and PMOS enhancement devices are used to construct all the logic circuits of the CMOS WFTA chips. Two basic circuits used are the inverter and transmission gate. Both circuits are built in a manner to insure that a "strong" signal is always output. This is accomplished by insuring that "1" is passed by a PMOS device, and a "0" is passed by an NMOS device. The inverter and transmission gate circuits are shown in Figure I-4.

The foremost attribute of the inverter circuit is that there is no direct-current flow when the inverter is not changing states, and, therefore, no power dissipation. The inverter outputs Vdd when a "0" is input and outputs Vss when a "1" is input. In either case, one of the MOS devices is switched off. Thus, no conducting path from Vdd to Vss is available, and no power is drawn except for a very small amount due to junction leakage.

The transmission gate circuit closely approximates an ideal switch turned on when S is "1" and off when S is "0".

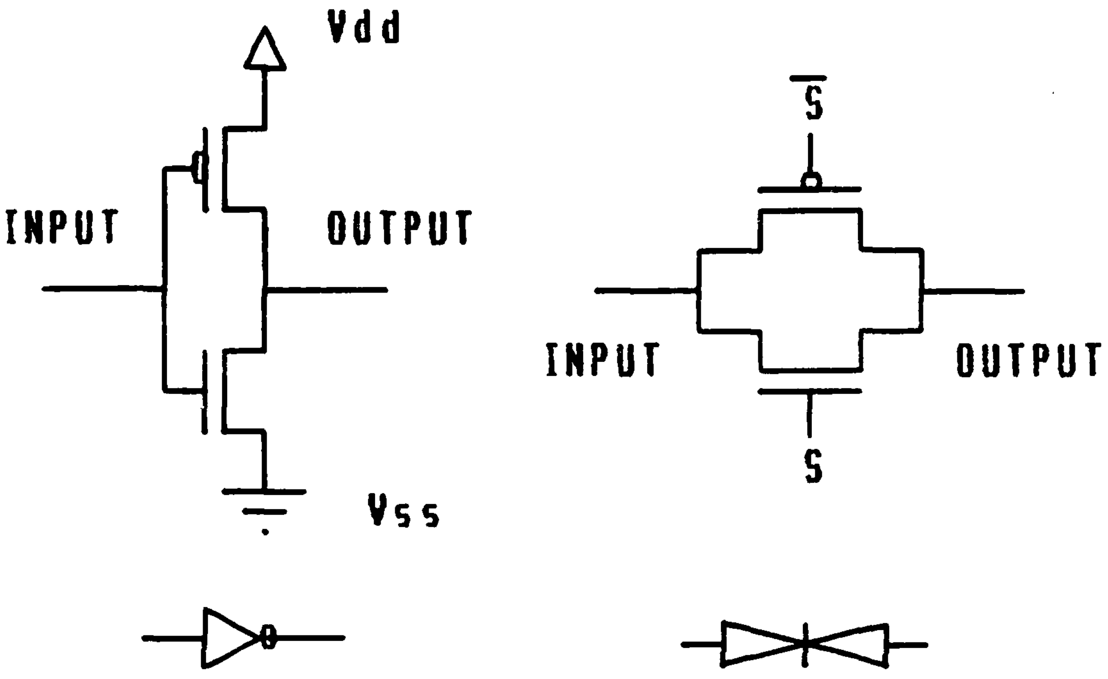


Figure I-4 CMOS Inverter and Transmission Gate.

DC Characteristic. The DC transfer characteristic for a CMOS inverter is depicted in Figure I-5. Notice that the CMOS inverter is usually designed to switch at an input voltage equal to half the supply voltage (Vdd/2). Also both transistors are on and current is drawn only while Vin passes from V_{tn} (threshold voltage for the NMOS device) to Vdd + V_{tp} (threshold voltage (-)for the PMOS device).

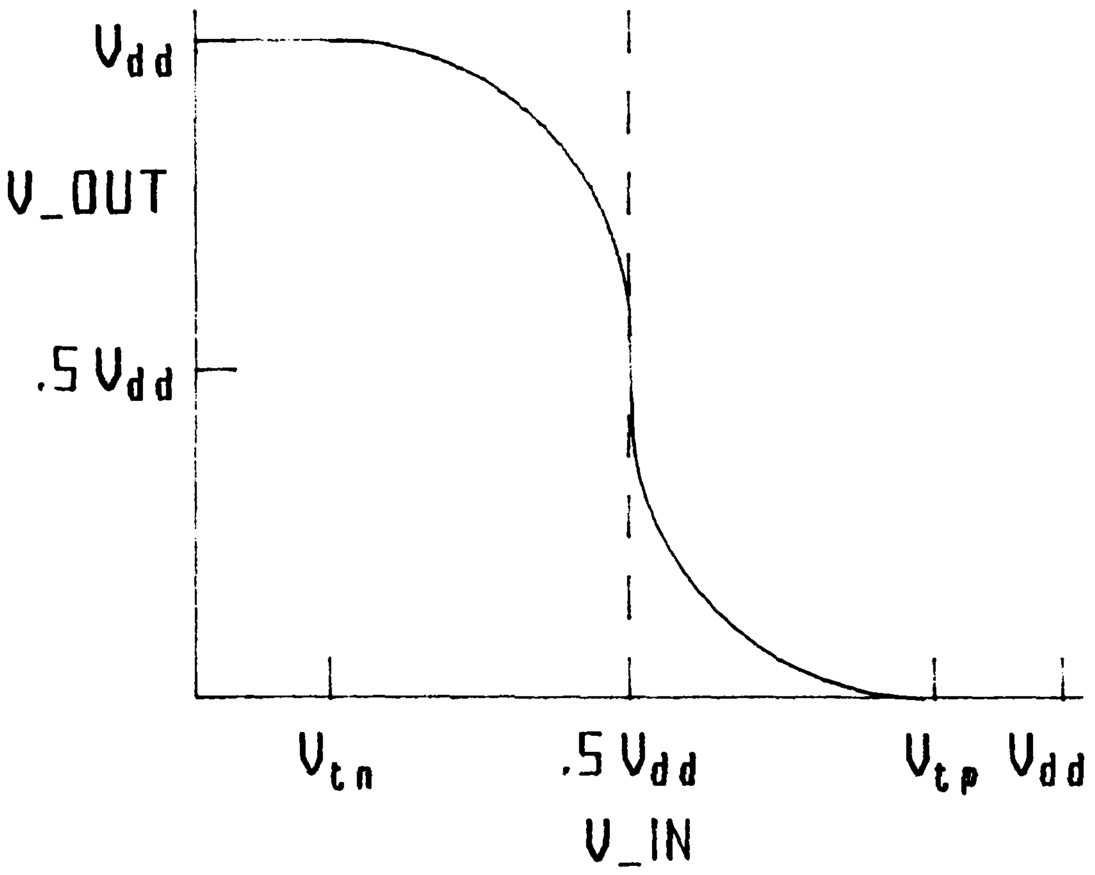


Figure I-5 CMOS Inverter DC Transfer Characteristic.

The DC transfer characteristics of the inverter depend on the ratio of the NMOS transistor gain (βn) to the PMOS transistor gain (βp). The transistor gains are determined by the effective surface mobility of the electrons in the device channel, permittivity of the gate insulator, thickness of the gate insulator, and physical dimensions of the channel. Of these four factors, the first three are determined by the fabrication process and the last is determined by the designer. Thus, by sizing the NMOS and PMOS transistor, the desired switching point for a CMOS inverter may be obtained. Figure I-6 shows the effect of varying the transistor gain ratio.

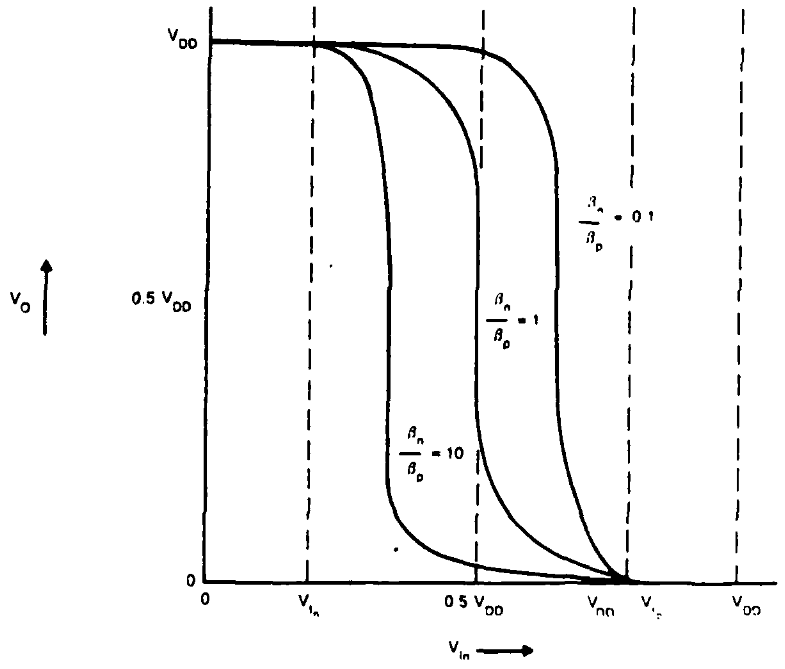


Figure I-6 Influence of $\beta n/\beta p$ (Weste and Eshraghian, 1985:50).

For good noise immunity, the majority of inverter's in the WFTA circuits were designed to switch at Vdd/2. DC SPICE models indicated that the ratio of NMOS to PMOS device widths should be 1:2 in order to realize the Vdd/2 switching point for the CMOS process used in the WFTA design. This 2:1 transistor ratio was used in most of the circuit designs to improve noise immunity and also to insure a symmetric waveform with respect to transient rise and fall times.

The output characteristics of a transmission gate are in Figure I-7.

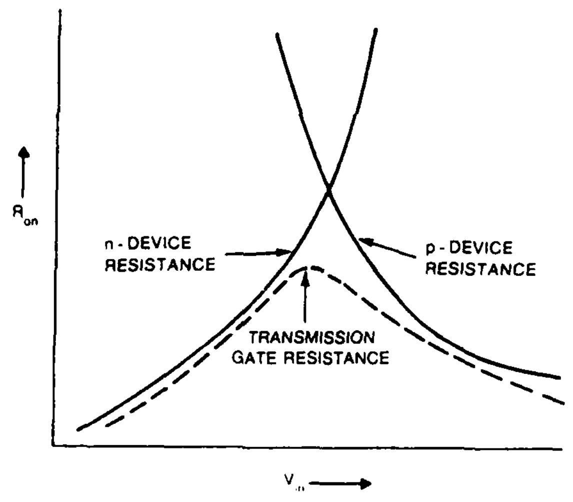


Figure I-7 Transmission Gate Characteristics (Weste and Eshraghian, 1985:57).

The parallel combination of the NMOS an PMOS resistances result in a transmission gate resistance that is low for passing low or high voltages (Vin).

Switching Performance (Lewis, 1983:6-8). For a CMOS inverter with $\beta n = \beta p$, the propagation delay for a high-to-low transition will equal that of a low-to-high transition. The switching delay, D, of that CMOS inverter will be determined by the devices' internal time delay, t, the supply voltage, Vdd, the transistor gain, β, and the capacitive load, C in the following manner:

$$D = 2\ C/\beta\ Vdd + t.$$

If the negligible internal device delay is ignored, and the supply voltage is assumed to be held constant, the CMOS delay is determined by the capacitive load and the size of the inverter devices (that affects the gain, β).

Thus, circuits requiring fast switching times must have the capacitive load held to a minimum, and the NMOS and PMOS devices sized to drive the capacitive load that does exist. The parasitic capacitances and, sometimes more importantly, the interconnect capacitances must be taken into account when designing any CMOS circuit that is under a speed constraint. Not only can the single inverter be sized to drive the capacitive loads, but a number of inverters can be staged up in size to drive large loads

Many factors must be considered when using successive stages, but a minimum speed staging requires each stage to be larger than the previous one by a factor of e = 2.7 (Mead and Conway, 1980:13).

CMOS Latch-up. CMOS circuits are susceptible to latch-up because of the presence of a p-n-p-n structure. During latch-up, the CMOS circuit presents a near short-circuit condition across the power supply. If preventive measures are not employed to limit the current flow, some metal or diffusion current paths may be permanently damaged. The four-layer structure that is susceptible to latch-up is shown in Figure I-8. Under normal biasing conditions, all junctions are reversed-biased. If, however, one of the source or drain junctions become forward-biased (due to momentary voltage transients at the input/output leads), internal gain amplifies the current until latch-up occurs.

The circuit designs and I/O pad designs of the WFTA chips are designed to protect against latch-up. Adherence to the design rules, numerous substrate contacts in every well, and avoiding structures that intertwine n- and p-devices were the rules followed in designing the WFTA circuits to prevent latch-up. The I/O pad circuits were designed with the protection diodes located as far away from active circuitry as possible and completely surrounded by guard rings. These techniques help to protect against

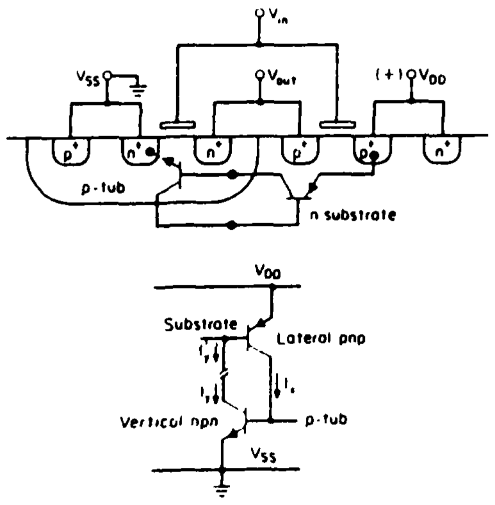


Figure I-8 p-n-p-n Structure (Ong, 1984:271).

latch-up in this most vulnerable area by preventing the diodes from supplying base current to other transistors during an electrostatic discharge. Figure I-9 shows the protection circuitry for the input pads and output pads. The output pad protection circuitry is minimal since the large output driver makes the output more resilent to voltage fluctuations.

Pseudo 2-Phase Clocking. The WFTA chips employ a clocking strategy that uses 2-phase nonoverlapping clock signals and their complements. This clocking strategy eliminates the possibility of race conditions that might exist in a design using a single clock and its inverse.

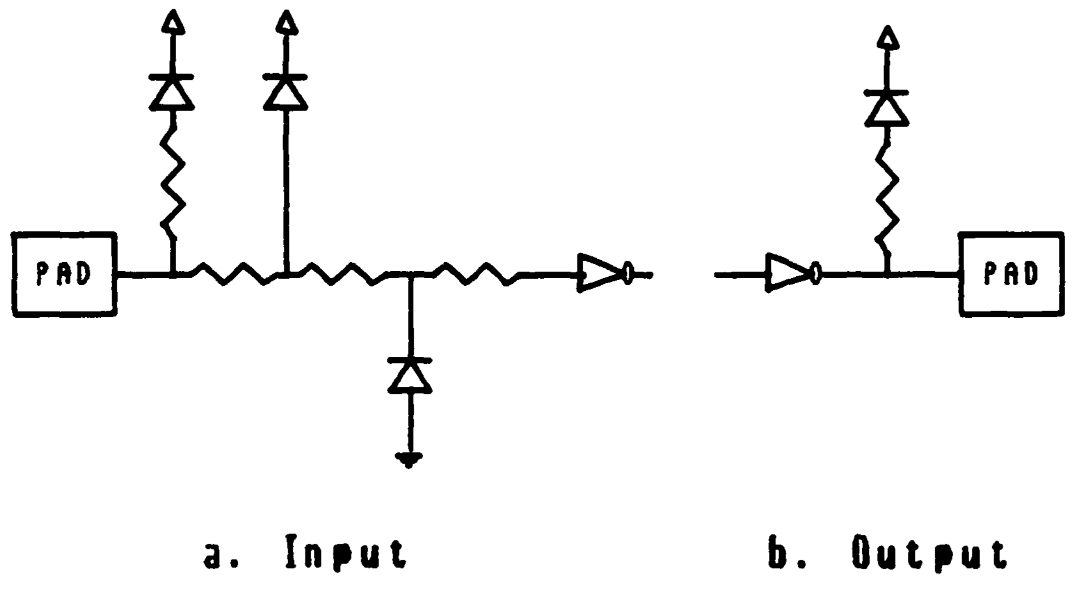


Figure I-9 Electrostatic protection circuitry.

Thus, there is a phi 1, phi 2, phi 1', and phi 2' required for clocked circuits on the chip. Usually, two master phi 1 and phi 2 clocks are distributed to local buffers that generate the inverses. The clock signals (phi 1, phi 2) are driven from off-chip to insure rapid transitions and sharp clock edges. The final chips will have clock pins at each corner of the chip to decrease the total distance (and resistance) through which the clock signals travel. This approach, coupled with properly designed local clock signal inverters, can help minimize clock skew and allows the circuit to function properly at high frequencies.

Approach

The responsibilities for major subsections of the WFTA processor chips and PFA pipeline architecture were distributed between the four thesis efforts presented previously. All efforts were directed toward implementation of a functional WFTA16 processor. The design of the WFTA15 and WFTA17 were to be accomplished in parallel as time permitted. The general WFTA16 floor plan provided general guidelines for the area restrictions on the functional blocks. This floor plan is depicted in Figure I-10.

The control signal requirements for the arithmetic circuitry, address generation circuitry, and chip interface circuitry were to be developed and design implementations selected. The WFTA16 subcircuit modules were to be submitted for fabrication and test in order to insure the proper functionality of each part of the chip design.

Sequence of the Presentation

Chapter II examines various methods of control circuit implementation that satisfy the requirements for the arithmetic control, address generation control, and chip interface control circuitry. For each circuit, a configuration that best satisfies the requirements is proposed for use in that circuit design.

Chapter III presents the arithmetic and address generation control circuits. Each circuits' method of

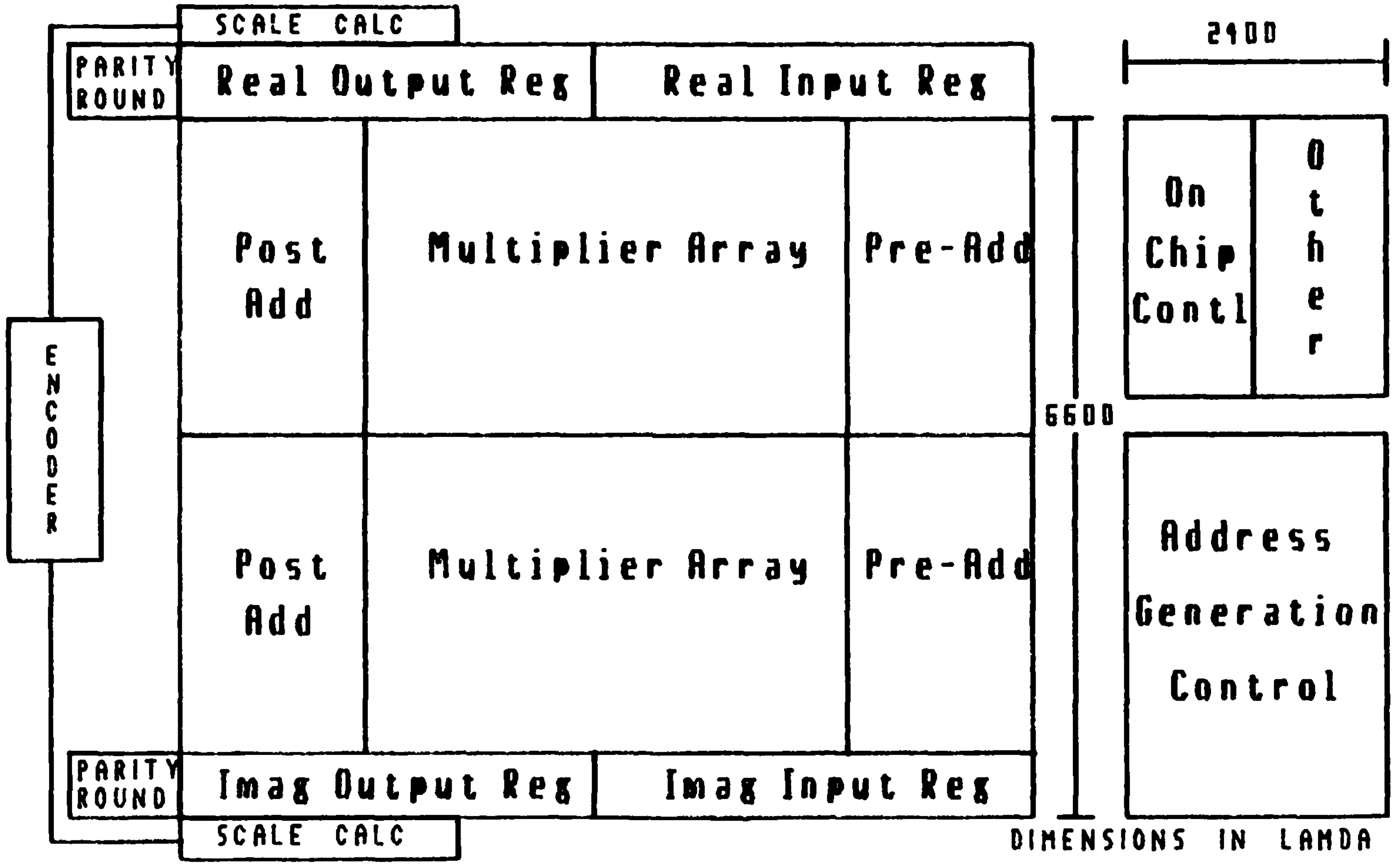


Figure I-10 WFTA16 Floorplan

operation, cell designs, and simulation results is presented.

Chapter IV describes the development and implementation of software that produces an automatic layout description of a dense ROM that is optimized for a minimum number of devices and drains. The results and benefits of the optimization procedure are presented.

Chapter V describes the fabrication procedures, test results, and evaluation of the arithmetic and address generation control circuitry developed for the WFTA16 processor.

Chapter VI presents the conclusions and recommendations for the WFTA control circuitry and ROM optimization/ automatic layout software.

II. Detailed Analysis of the Problem

Arithmetic Control

Requirements. In this section, the signals required to control the arithmetic circuitry on the WFTA16 will be described. First, a brief explanation of the arithmetic circuitry will be given. More detailed information on this circuitry can be obtained from Coutee (Coutee, 1985). Second, a description of the signals required to control the on-chip arithmetic circuitry will be presented.

Arithmetic Circuitry Overview. The circuitry that performs the Winograd DFTs is a bit-serial implementation of the pre-additions, multiplications, and post-additions of the WFTA. These operations are performed in word-parallel fashion on 16 streams of complex serial input data. Data I/O transfers to and from I/O shift registers are performed in parallel. The I/O transfers are performed in parallel since the processor's throughput is limited by the time required for the data I/O. A number of numerical precision and error detection operations are performed on the real and imaginary data as they serially pass certain points in the architecture. Figure II-1 presents the overall block diagram of the real or imaginary arithmetic circuitry for WFTA16.

Arithmetic Component Description. The Watchdog (W/D) circuitry functions to provide an error checking

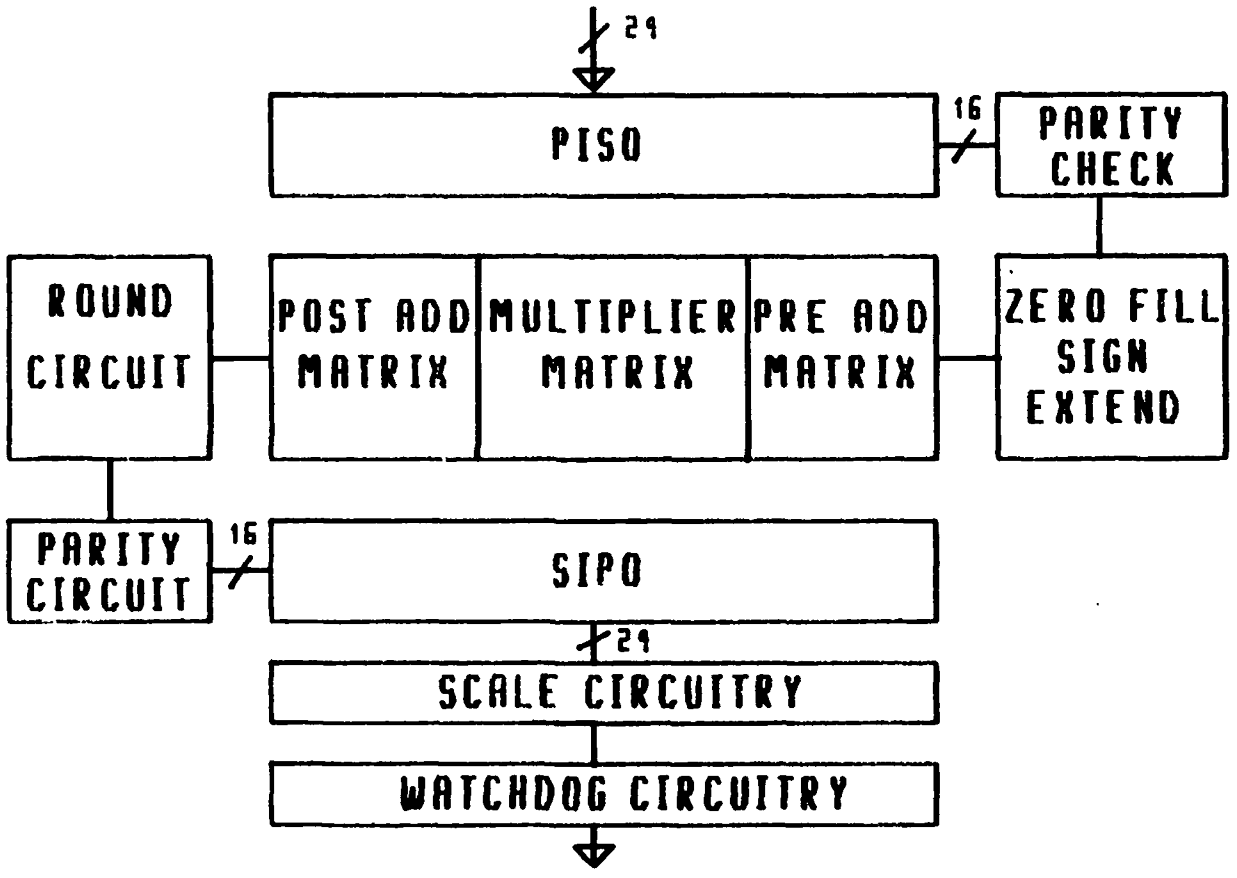


Figure II-1 WFTA16 Arithmetic Circuitry

capability which monitors another WFTA chip. The input addresses and the output data and addresses of a WFTA chip in W/D mode are continuously compared to those of an active WFTA processing chip. Any discrepancies in the data or address values cause the W/D error flag to be set for the duration of the transform.

The Parallel-In Serial-Out (PISO) circuitry and Serial-In Parallel-Out (SIPO) circuitry convert parallel data flow to serial and serial data flow to parallel respectively. The PISO and SIPO are both capable of shifting data in and out simultaneously using only one clock cycle to exchange an entire block of 16 data words from parallel registers to serial registers (or vice versa).

The Parity Checking circuitry examines the data and parity bits of each data word to check for input data errors. If any parity errors are found, a parity error flag is set for the duration of the transform.

The Zero Fill/Sign Extension circuitry appends sign extensions or zeros to the appropriate bit positions of the data word depending on the scaling factor input to the chip. This circuit extends the 23-bit data representation used externally to the 32 bits per data word that are used to perform the calculations within the arithmetic hardware.

The Pre-add, Multiplier, and Post-add circuits implement the WFTA algorithm as described in Chapter I.

The Rounding Module rounds the 32-bit internal result to a 23-bit result before output. The Parity Module recalculates the odd-parity for each data word result and appends that parity to the twenty-fourth bit position.

Finally, the scaling circuitry keeps track of the largest result to inform the next WFTA chip or the host of the scale factor to be used in subsequent calculations. The scaling is used to increase the numerical precision of the final result. A more detailed explanation of the scaling procedure can be found in (Taylor, 1985).

Control Signal Description. Remarkably few control signals are required to coordinate the operation of all the arithmetic circuitry described above. The WFTA16's data throughput is limited by the I/O data rate. Since two clock cycles are required to input/output each data word of the sixteen that can be stored in the SIPO and PISO, 32 clock cycles are required for each data block. This figure of 32 determines both the number of bits in the internal data word representation and the number of clock cycles before most of the control signals are repeated. After the WFTA16's internal pipeline is filled, the same control signal pattern will be repeated every 32 clock cycles until the transform is completed.

In the following pages, a brief description of each control signal function and the signal's waveform type will be presented. These descriptions should give a general understanding of the control signal requirements. Each control signal description will refer to one of the waveform types shown in Figure II-2. The waveform types will be referred to by the letter designator given in the figure; A for a divide-by-two/half-frequency clock signal, B for a repeating interval signal, C for a repeating pulse signal, D for a non-repeating master reset pulse signal, and E for a signal whose waveform is a function of the scale factor. Letter designators may be combined (such as D-E for a repeating interval signal that rises or falls as a function of the scale factor) to indicate that the waveform has more than one characteristic.

The control signals are listed below by function:

1. W/D:

   (a) Input Check - Enables comparators when the data is stable on input lines.
   Waveform: A

   (b) Output Check - Enables comparators when the data and addresses are stable on output lines.
   Waveform: A

   (c) RESET - Resets the W/D error flag at the start of each transform.
   Waveform: D

2. PISO:

   (a) Shift Down - Loads the 16 complex data words into the PISO, one data word every other clock cycle. The PISO is 16 registers deep.
   Waveform: A

   (b) Latch - Moves the 16 complex data words shifted into the PISO to the 16 registers used to serially shift the data out. The Latch signal must occur for one clock cycle when the PISO is not shifting down or right.
   Waveform: C

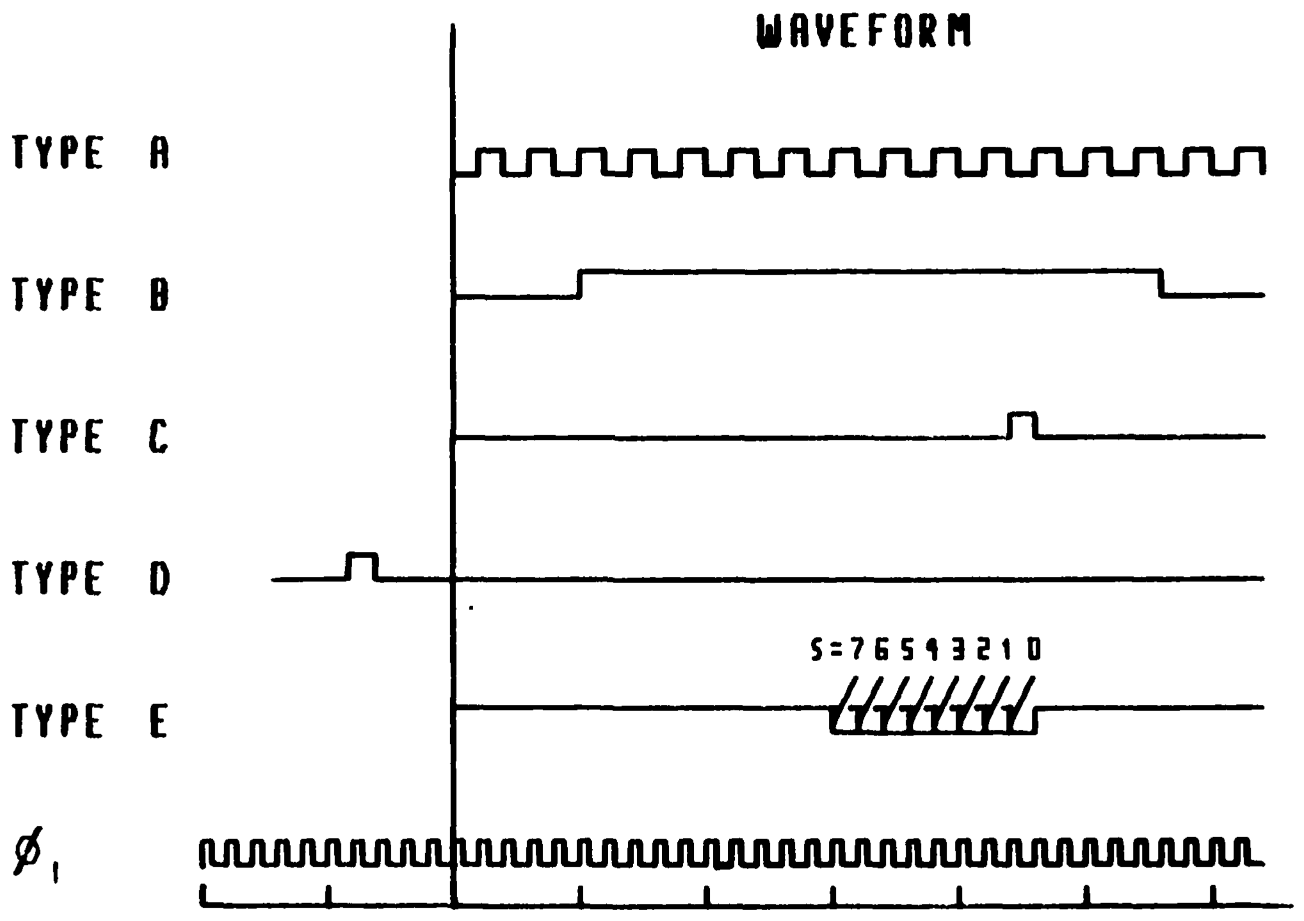


Figure II-2 Control Signal Waveform Types

(c) Shift Right - Interval signal that shifts the 24 bits of each of the 16 data words out of the PISO toward the pre-add matrix.
Waveform: B-E

3. Parity Checking Circuit:

(a) Check - When this signal is high the parity of all bits in the 16 data words is checked as the bits are shifted out of the PISO. The calculated parity is compared to the parity bit for each word. An error signal goes high if the parity of the word is not odd.
Waveform: B-E

(b) Check Reset - This pulse signal resets the parity checking circuitry after each group of data words has been checked for parity.
Waveform: C

(c) Latch - This signal latches any parity error found on the 16 data words into the parity error set/reset flip-flop. This pulse signal occurs everytime valid results exist on the parity check result line (i.e. 24 clock cycles after the start of shift right PISO).
Waveform: C-E

(d) Error Reset - This signal is used to reset the parity error flag at the start of each transform.
Waveform: D

4. Zero Fill/Sign Extension Circuit:

(a) Zero Fill - When high (active), zeros are shifted into the arithmetic circuitry to scale the input data up to 32 bits.
Waveform: B-E

(b) Sign Extension - When high (active), sign extensions are shifted into the arithmetic circuitry after the MSB of the data. All data words have at least 5 sign extensions.
Waveform: B-E

5. Pre-add, Multiplier, and Post-Add Circuits:

(a) Carry Reset - This pulse signal propagates through a shift register, resetting the carry cell of each adder or multiplier as it goes. The carry cell is reset just before the LSB of the next set of 16 data words arrives at each adder or multiplier.
Waveform: C

(b) Multiplier Round - This pulse signal rounds (rather than truncates) the result of the multiplier to 32 bits before reaching the post-add matrix.
Waveform: C

6. Rounding Circuitry:

(a) Round Calculate - When active, this signal allows the Rounding circuitry to round the output data word by adding the most significant bit not kept to the output data word. Since bits 9 to 31 of the 32 bit result are used, bit 8 is added to bit 9 and any carry is forwarded.
Waveform: B

7. Parity Circuitry:

(a) Calculate - When active, this interval signal commands the parity cell to calculate odd parity on the serial bits being output. The parity is calculated on the 23 bits that are kept and output to the SIPO.
Waveform: B

(b) Append - This pulse signal causes the parity cell to append the calculated parity to the 24th bit position of each of the 16 real and imaginary output data words.
Waveform: C

8. SIPO:

(a) Shift Down - Outputs the 16 complex data words from the SIPO, one data word every other clock cycle. The SIPO (like the PISO) is 16 registers deep.
Waveform: A

(b) Latch - Moves the 16 data words that were

serially shifted into one set of registers to the 16 registers used to output the data in parallel. The Latch signal must occur for one clock cycle when SIPO is not shifting down or right.
Waveform: C

(c) Shift Right - Interval signal that shifts the 24 bits output from the arithmetic circuitry into the 16 registers of the SIPO that receive serial data.
Waveform: B

9. Scaling Circuitry:

(a) Update - This signal (when active) allows the scaling circuitry to compare the stable data on the output bus to the current highest value seen. If the output data is of higher value, its sign bit location is remembered.
Waveform: A

(b) Reset - This signal resets the scaling data word to zero before the next transform data results are output.
Waveform: D

Approaches, Tradeoffs, and Solution.

Approaches. The WFTA16 is a sequential state machine with arithmetic control signals as described above. A method of implementing the control circuitry to generate the given control signals is needed. There are a number of design methods available to implement a sequential control circuit. The control for a dedicated signal processor such as the WFTA requires only limited programmability allowing for the use of a faster, less flexible hardware design. The most popular of these control methods include custom logic implementations, use of gate arrays or programmable logic arrays (PLAs), and microprogramming.

Custom logic implementations are those methods that rely on decision logic modules built from individual gates and flip-flops. The design can be obtained from design methods that use state and excitation tables, methods that use one flip-flop per state variable coupled with a decision logic module, or methods that use a counter/decoder and decision logic module (Mano, 1979:410). The common element of all design methods for a custom logic implementation is that external inputs and present state inputs feed into a decision logic module built from custom combinational logic elements.

Gate arrays can be used to implement the entire control circuitry or to replace the custom decision logic module for the custom logic implementations. Gate arrays are fixed arrays of identical logic circuits. Designing with gate arrays consists of specifying the wiring interconnections between the given logic elements in the array (Ong, 1984:329).

A PLA implementation of the control circuitry would require a sequence register or counter coupled with a PLA. The sequence register would keep track of the present state of the machine while the PLA would use the present state and external inputs to generate output control signals (Mano,1979:413). A slight variation to the normal use of a PLA can be used when designing the control circuitry. If the sequence register's outputs can be tied to the product terms of the PLA so that only the one (or a few) product term that corresponds to the present clock cycle within the timing diagram is allowed to be high at any one time, the speed, power dissipation, size, and complexity of the AND-plane can be greatly reduced (Linderman, 1984:90).

Finally, a microprogramming approach can be used to implement the control circuitry. Strings of 1's and 0's comprising control words are stored in a control memory (often a ROM). There are as many control words as there are different possible control signal variations. The problem then becomes one of controlling the address register that addresses the control memory (Mano, 1979:414). If the memory can be accessed in the proper order, each bit of the control words could be used to implement a control signal.

Trade-offs. Each method of implementing the control circuitry has its advantages and disadvantages. The goal is to use the implementation that will provide the WFTA16 with the best speed, flexibility, and simplicity in a minimum amount of area.

The control signals must be switched at rates of over 50 MHz (for the 3 micron process) in order to keep pace with the bit-serial arithmetic circuitry. All of the implementations can operate at this speed with exception of the microprogramming approach. It would take a great deal of effort to make the microprogramming approach work for this application since the control ROM would have to be very small in order to obtain the required access speed.

Since the WFTA chips are being designed from the ground up, the control signals may change a number of times before the final timing diagram is achieved. Because of this, there is a great need for the design to be easily changed. This is not the case with any of the custom logic implementations, and, therefore, they must be ruled out.

Of the two remaining options, a sequence register and PLA or gate arrays, the PLA approach offers the better use of area and is simpler to design and redesign. Because the PLA approach offers a simple and flexible implementation that can operate to speed in a small area, it will be used to generate the control signals for the WFTA16.

Solution. Figure II-3 depicts the type of PLA

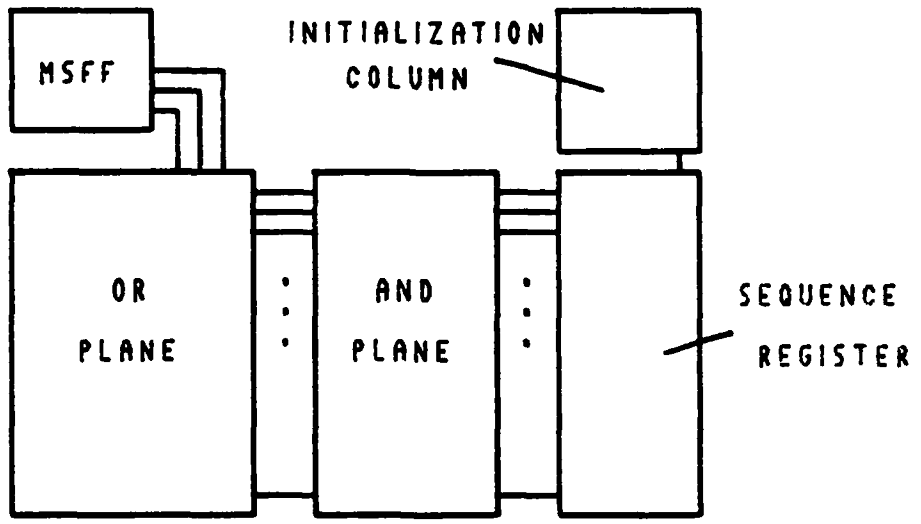


Figure II-3 General PLA Controller

implementation to be employed to create the required control signals. This implementation offers better speed and power dissipation than one that would have the sequence register's outputs fed into the AND-plane inputs.

Improved power dissipation results from driving only the product term lines that correspond to the present clock cycle of the timing diagram high at any one time. The speed is better for two reasons. First, the low power dissipation allows the PLA devices to have much higher current drive. The larger array devices are capable of switching the PLA lines quicker than the smaller devices that would have to be used in a standard PLA in order to keep its power consumption down to acceptable levels. Second, the signal propagation distance is smaller in the proposed PLA implementation, thereby decreasing the time required to generate the output signals. The AND-plane inputs of the WFTA will be stable long before they are needed and will remain unchanged throughout the transform (The inputs consist of only a three-bit scale factor that is loaded prior to the start of the DFT). Thus, when the product terms are raised by the sequence register, the AND-plane inputs will already be stable, and the delay down the AND-plane will be eliminated.

The final decision for the controller regards the type of present state sequencer that will be used to output the single pulse for each clock cycle in the timing diagram. A new present state sequencer that is simple, fast and requires a minimum amount of area is desired. Again, there are a number of possible solutions that can implement the sequencer. They are a counter/decoder, ring counter, or Johnson counter (Mano, 1979:286).

The counter/decoder method uses a counter that cycles through the required number of distinct states while a decoder decodes the states of the counter into a sequence of pulses. The ring counter is a circular shift register with only one flip-flop being set at any particular time. A single bit is shifted from one flip-flop to the next to produce the sequence of timing pulses. The Johnson counter is a special ring counter that uses only 1/2 the number of flip-flops to achieve the same number of timing signals. It accomplishes this by passing more than one bit down the flip-flop chain and using 2-input AND-gates to output the proper sequence of timing pulses.

Since all three ways of implementing the PLA's sequencer are of comparable complexity, the choice will be based upon the speed of the circuit and area needed to construct it. The slowest method of implementing the sequencer is the counter/decoder method. This method requires the signal to propagate through all flip-flops of the counter in the worse case as well as through the decoder. The ring counter and Johnson counter are similar in speed performance, although the Johnson counter is slower due to the extra level of 2-input AND-gates required on the outputs. The ring counter is the fastest method of the three. Additionally, since a ring counter can be layed out in a dense array with one flip-flop adjacent to each of two PLA product terms, only a small area along the side of the PLA is needed for the counter's layout. The area required for the counter/decoder, or Johnson counter, added to the area required for routing to the product term lines would be at least as large as the area required for the ring counter, if not more. Therefore, the ring counter implementation of the present state sequencer will be used to drive the control PLA.

Configuration of the Control Sequencer. Thus, the proposed final configuration for the arithmetic control circuitry is a Control Sequencer comprised of a PLA, ring counter with outputs driving the product terms of the PLA, and control signal output flip-flops. The proposed configuration is depicted in Figure II-4. The ring counter will continually cycle a bit around the counter. The pulse will drive each of the PLA product terms high as it travels down the ring counter. When a particular set of product terms are driven high by the ring counter, it may stay high (if all the AND-plane conditions are met) and may cause a control pulse to be output from the OR-plane. The control pulse may set or reset a Set/Reset flip-flop (SRFF) or be broadcast to the arithmetic circuitry through a master/slave

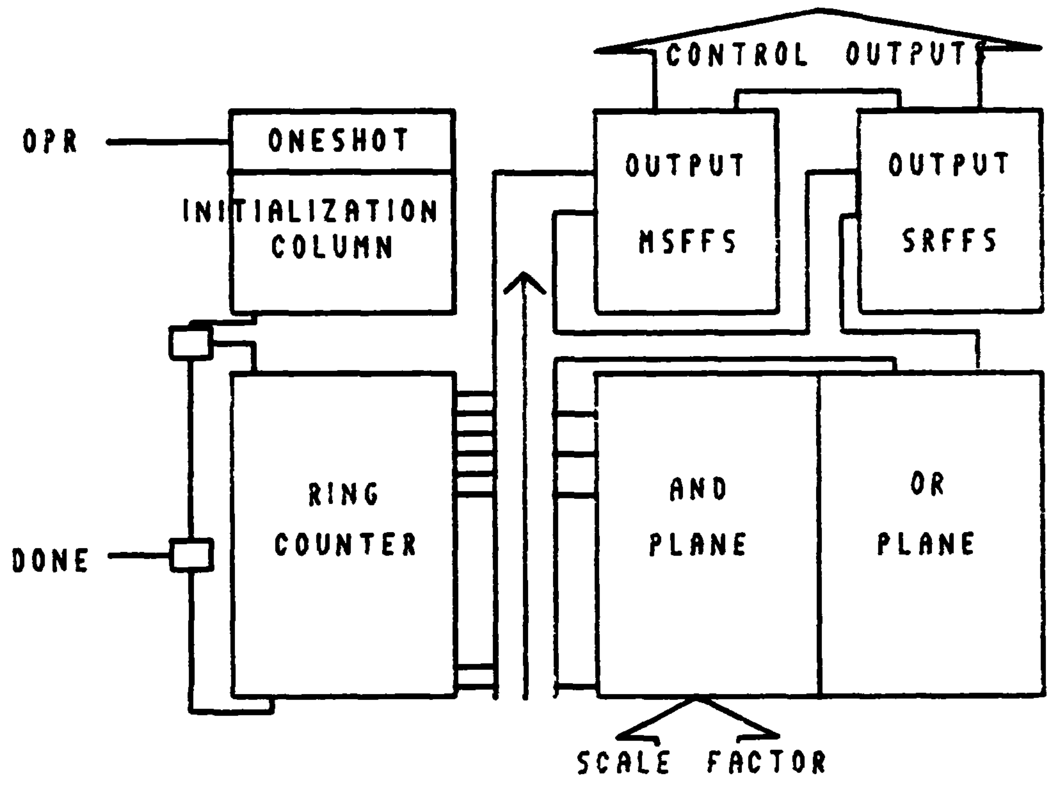


Figure II-4 Proposed Control Sequencer.

flip-flop (MSFF). Since the control signals depend only upon the present state and the scale factor, the only external AND-plane input will be the scale factor number. If a control pulse is always required at a particular time regardless of the state of external inputs to the AND-plane (the scale factor), the signal may be taken directly from the ring counter without going through the PLA.

The Control Sequencer must be started and stopped for each transform the WFTA16 performs. To start the Control Sequencer, it must have an initialization column with a one-shot. The one-shot generates an output pulse when the OPERATE signal transitions from low to high. In order to stop the Control Sequencer, the bit that cycles through the ring counter must be prevented from returning to the first MSFF in the chain. To reinitialize the Control Sequencer, all ring counter and output flip-flops must be resettable. They must reset when the OPERATE signal is driven low again (see the Chip Interface Control section of this chapter).

Address Generation Control

Requirements. As was described in the first chapter, the WFTA makes use of fixed coefficients which result in fewer I/O transfers. However, the transform data I/O must still be accomplished for each WFTA chip in the pipeline. This I/O transfer between the WFTA chip and the external RAM must take place rapidly and at precisely the proper instant to insure accurate transform results. The data I/O transfers are further complicated by the PFA which uses the Good and Thomas prime factor index in mapping one-dimensional DFT problems into a multi-dimensional DFT and back to a one-dimensional result (Good, 1971; McClellan and Rader, 1979). That is, the transform data must be input and output in a special order. The retrieval and output of the transform data will not be a matter of simply incrementing a data address counter. The input data will be spread about the input RAM according to the Chinese Remainder Theorem for the size transform being performed. The addressing sequence must follow the ordering of the Chinese Remainder Theorem to input and output the data. A more detailed discussion of the Chinese Remainder Theorem and its use in the PFA is given in Taylor (Taylor, 1985). Thus, some special address generation control circuitry is needed to generate input and output addresses as required by the PFA's data shuffling scheme.

Not only is the address generation control circuitry required to produce the proper input and output addressing sequence for each WFTA chip and each transform size, but it must be built using the minimum possible area of the WFTA chip, and operate at speeds over 25 MHz. The circuitry must operate at 25 MHz (3 micron) in order to drive the input and output address buses with a new address every other clock cycle at the target operating speed. Since each WFTA chip in the pipeline (the 15, 16, and 17 point chips) will have four different DFT sizes to calculate, 12 variations of the address generation control circuitry must eventually be realized. Therefore, the flexibility and ease of implementation for the circuit becomes an important constraint.

Each WFTA chip's address generation control circuitry must be capable of generating approximately 4.5K 12-bit addresses in the proper order. 12 bits are required per address in order to address all the data words in the 4080 point transform. The approximately 4.5K addresses that must be generated consist of the sequence for the 4080 point transform plus three other transform size sequences which vary for each WFTA chip (see Chapter III).

Approaches, Trade-offs, and Solution.

Approaches. Two methods to implement the address generation control circuitry were examined. The address generation could be accomplished through special purpose, custom-built circuitry made up of registers, parallel adders, parallel subtractors, comparators, and gates. Each implementation would be designed specifically for a particular WFTA chip. The circuitry would implement the Chinese Remainder Theorem for each of four DFT sizes per chip. The custom-built address generation hardware is shown in block diagram form in Figure II-5.

The address generation control circuitry could also be implemented using a counter and a ROM. The ROM could be personalized with the I/O data addresses in the correct sequence, and the counter could then sequentially address the ROM in order to place the proper shuffled data addresses on the I/O address bus. The four different DFT sizes could be accommodated by starting the counter at four distinct locations of the ROM.

Trade-offs. There are five main factors to consider in determining which of the two methods to use to implement

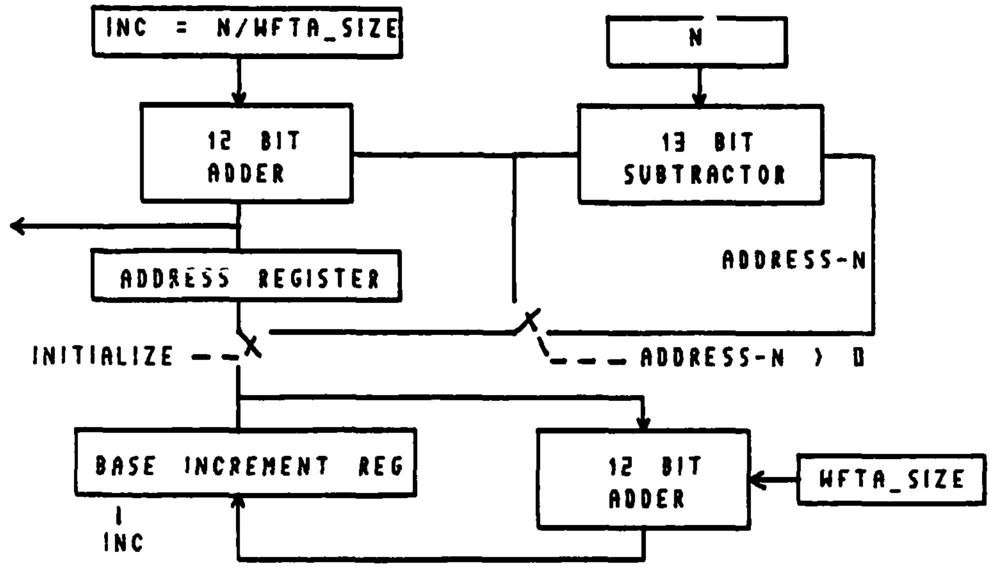


Figure II-5 Custom-Built Address Generating Hardware (Linderman, 1983:196).

the address generation control circuitry. These five areas are speed of operation, complexity of design, flexibility, area required, and extensibility of the solution.

The custom-built hardware had a distinct advantage over the ROM in the amount of silicon area required. The custom-built hardware could be implemented in 1/2 the area it would take to build the ROM circuitry.

The two methods of implementation would generally have a comparable speed of operation and complexity of design. The speed of the custom-built hardware is slowed by its need for numerous parallel adder, subtractor, comparator, and decision modules to calculate the addresses. The ROM implementation is hindered by its need to retrieve each data address from the ROM array and the operation of the counter that addresses the ROM.

The complexity of the custom-built hardware stems from the difficulty in calculating the correct sequence of data addresses for each WFTA chip and each DFT size. The complexity of the ROM stems from the requirement to personalize it with over 50,000 bits (although this complexity can be significantly reduced by developing software to perform the ROM personalization automatically).

The ROM implementation possesses an advantage over the custom-built hardware in its flexibility and extensibility. Not only could the same basic ROM design be used on all three WFTA chips, but it could also be used on any other CMOS project that required a ROM.

The last two advantages of the ROM described above are very significant in the current context of CMOS design at the Air Force Institute of Technology (AFIT). Since CMOS design is just getting its start at AFIT, a ROM cell library is needed. The ROM implementation could be extensible to most any other future project especially microcode stores for

Reduced Instruction Set Computers (RISCs). Design time would also be saved for the WFTA15 and WFTA17 chips if the ROM implementation was used. Rather than redesigning the custom hardware, new addresses could simply (and automatically via personalization software) be placed in the same ROM design.

Solution. The ROM implementation appears to be more desirable than the custom-built hardware if there will be enough room on the WFTA chips. Fortunately, there is enough room in the control section of the WFTA chips to support the space required to implement the ROM address generation circuitry because a very dense ROM design will be used. This dense ROM is one that uses a very compact XROM cell (Wilson and Schroeder, 1978; McKenny, 1980). The XROM cell is capable of storing four bits in a cell that is just under 8 x 8 microns square (using the design rules and 1.25 micron process given in Appendix A).

More improvement can be obtained from the XROM in the area of speed if precharging and special output sense amplifiers are used. These two improvements further the desirability of the XROM implementation. Additional gains in speed can be made by outputting more than one data address from the XROM at a time. By making the XROM word width 2, 4, or 8 times the width of a single data address, a similar factor gain in output can be achieved.

In either implementation of the address generation control circuitry, both input and output data addresses must be generated. Once the internal serial data pipeline of the WFTA chip becomes filled, both input and output addresses must be generated simultaneously. There are a number of ways to generate the input and output addresses. One method is to have two XROM circuits, one for the input addresses and one for the output addresses. This approach is not required, however, since the data words will be output to the same address location as they were input. Therefore, two different approaches can be considered.

One approach is to delay the input data addresses the proper number of clock cycles in an array of shift registers until the data word passes through the arithmetic circuitry and is ready to be output. This approach, however, uses an excessive amount of area for the array of shift registers. In fact, for the WFTA16, the necessary delay would require a shift register array larger than the size of another XROM. The final approach is to clock the XROM at a rate twice what is needed for just the input data addresses or just the output data addresses, and retrieve both input and output addresses from the same XROM. The only additional circuitry required would be a constant number subtractor to be used to retrieve the output data address which is a constant number of addresses "back" in the XROM. The input data addresses would be output from the XROM into a set of flip-flops, and then the output data addresses would be output to another set of flip-flops. Both accesses would be accomplished in the necessary time to keep both input and output data addresses available for external use.

The overall XROM access throughput can be increased as required by designing it to read out a larger number of addresses at a time. Given the final speed of the XROM circuit and the rate at which both the data addresses must be presented (25 MHz for 3 micron), the width of the XROM can be directly calculated. The width of the XROM is important to its final speed, however, and a number of design trade-offs exist to insure the final XROM will be able to operate at speed (although many sacrifice area and circuitry to obtain the greater speed).

Configuration for the Address Generation Control Circuitry.

Circuit Description.

The final configuration for the address generation control circuitry is a counter/subtractor circuit used to address an XROM that will store all the sequences of input and output data addresses. Figure II-6 depicts the proposed address generation control circuitry. The XROM will output a multiple number of data addresses on each access in order to achieve the required speed of operation. The XROM may be split vertically in order to allow more wordline drivers to drive the XROM faster. Precharging and output sense amplifiers will also be used to increase speed.

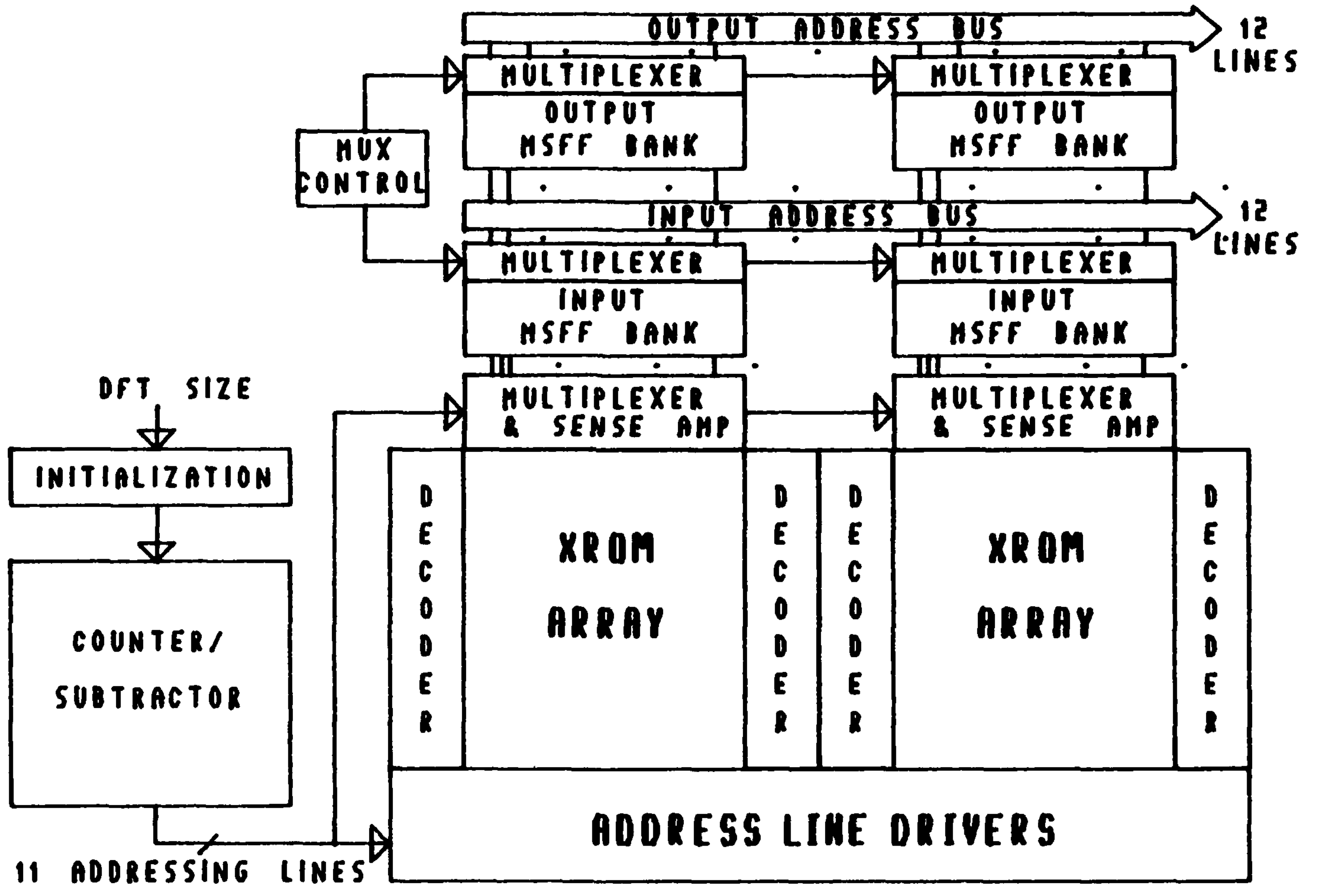


Figure II-6 Proposed Address Generation Circuitry

The counter/subtractor circuit will be used to address the XROM sequentially for both the input and output addresses. The XROM counter will start at a value determined by the DFT size to be performed, and continue addressing the XROM to retrieve input data addresses. The subtractor will subtract a constant from the counter value and use that result to address the XROM. This addressing will retrieve the output data addresses that will lag a constant amount behind the input data addresses.

The contents of the XROM that are output on an input data access will be shifted into a bank of flip-flops for input data addresses. The contents of the XROM that are output on an output data access will be shifted into a bank of flip-flops for output data addresses. Both banks of flip-flops will be multiplexed in order to output one address at a time on the output data address bus and the input data address bus.

The XROM will provide the WFTA chip design effort with a great deal of flexibility in changing address sequences if required. Additionally, the same design can be used on all three WFTA chips with differing XROM personalizations. The XROM cell will also provide a great savings in area of the chip used over standard ROM designs.

Control Signals Required. The XROM address generation circuitry and the associated data I/O transfers will require some control signals. The control signals will be generated by the Control Sequencer in the same manner as it generates the arithmetic control signals. Each of the eleven control signals necessary to control the operation of the XROM circuitry will be described below. The description will be presented in the same manner as was done for the arithmetic control signals. The same waveform letters will be given from Figure II-2. The signals required by the address generation control circuitry are:

1.	Precharge	-	This signal (inverted sense) is used to allow faster access to addresses stored in the XROM. It is used to precharge the bit and sense lines high. Waveform: B
2.	Load	-	This pulse is used to load the XROM addressing counter with the proper value for the size DFT being calculated. Waveform: D
3.	Increment	-	This pulse signal causes the XROM addressing counter to be incremented each time it goes high. Waveform: C
4.	Latch	-	This signal causes the contents of the XROM addressing counter to be latched into the buffer register. Waveform: C
5.	IN/OUT	-	This signal multiplexes the input data addressing value and the output data addressing value onto the XROM's decoder address lines. Waveform: B
6.	Shift-Up-In	-	This pulse signal causes the current output of the XROM to be latched into the input data addresses' bank of

flip-flops. From here the input data addresses are multiplexed onto the input address bus.
Waveform: C

7. Shift-Up-Out - This signal causes the current output of the XROM to be latched into the output data addresses' bank of flip-flops. From here the output data addresses are multiplexed onto the output address bus.
Waveform: C

8. DONE-IN Compare - This interval signal informs the DONE-IN comparator circuit that a valid address is on the bus and a valid check can be formed to see if this is the last input address.
Waveform: C

9. DONE Compare - This signal performs the exact same function as DONE-IN compare except the check is for the last output address.
Waveform: C

10. Counter Clock - This two phase signal is a quarter frequency clock used as input to the counter circuit. The ripple carry of the counter requires a slower clocking speed.
Waveform: A

11. Write Strobe - This signal is driven off-chip to the output memory. This signal goes high one clock cycle after each output address has been driven onto the output address bus. The signal enables the memory write so the stable transform output data can be written into the output RAM.
Waveform: A

With these control signals, the XROM address generation control circuitry can perform its function of outputting the addresses for both the input and output transform data, and the output data can be successfully written to memory.

Chip Interface Control

This section, unlike the first two major sections of this chapter, addresses the requirements, approaches, and solutions for control that is required between WFTA chips or between a host processor and a WFTA chip. Although chip interface control is not the major thrust of this thesis effort, sufficient detail had to be known about how the WFTA chip was going to communicate with external elements in order to incorporate all necessary on-chip control circuitry.

Requirements. There are a limited number of communication signals that must be passed between the WFTA chip and the outside world in order for it to do useful, accurate, fault-tolerant, and precise work. These signal requirements or functions are listed below. Some signals are relatively straightforward and little controversy regarding their worth and form of implementation can be found. Others have a few possible implementations that could be considered. These interface signal descriptions will, however, present the type of implementation found to be the best with regard to function, minimization of pins needed, simplicity, and past experience (Linderman and others, 1985:762). The two major options available for signals that pertain to accessing the pipeline memories or the W/D operation mode will be presented. These two major options are off-chip interface control or on-chip interface control. The external WFTA chip control signal requirements are as follows:

1.	OPR	- The chip operate signal is an input signal and serves two functions. It instructs the WFTA chip to start the DFT calculation when it goes high. It resets the WFTA chip when it is returned to a low state.
2.	DONE	- The chip done signal is an output signal and informs the external processor that the WFTA chip has completed the DFT and has finished outputting all data to the output memory.
3.	W/D ERROR	- When this signal goes high and stays high at any time during the transform calculation, a WFTA chip in the watchdog mode of operation has detected an error committed by the active WFTA processor.
4.	PARITY ERROR	- When this signal goes high and stays high at any time during the transform calculation, a parity error on the incoming data word(s) has been seen.
5.	W/D Mode	- The chip Watchdog mode signal is an input from an external processor informing the WFTA chip that it is to operate in the watchdog mode. The output pins will be disabled, and the chip will compare its operation and results to the active WFTA chip.
6.	W/D START-UP	- This signal is required only if the ON-CHIP interface control method is used. The signal tells the WFTA chip in watchdog mode when to start its operation in order to stay in synchronous operation with the active WFTA chip. It is an output signal and is tied to the watchdog WFTA chip's OPR pin.
7.	DFT Size	- This signal requires two pins for the four possible DFT sizes each WFTA chip can perform. It is an input, and instructs the WFTA chip which size DFT is to be performed.

8.	Scale Factor	- This input signal requires three pins and informs the WFTA chip of the magnitude of input transform data.
9.	Scale Output	- This output signal requires three pins, and is used to tell the host processor or the next WFTA chip in a pipeline what the highest magnitude of output data was.
10.	Load State	- This input signal is used to multiplex two different type signals on the same input signal pin. The purpose of this multiplexing is to save on total number of pins.
11.	Pipelined Memory Access Signals	- These signals differ depending on whether ON-CHIP or OFF-CHIP interface control is used. If ON-CHIP circuitry handles the interface control, five signal pins are required for WFTA chip Hand-Shaking to control memory accesses. These five signals are:

a. Done with Input memory.
b. Toggle memory command.
c. Memory switched output.
d. Memory switched input.
e. Next memory available.

If OFF-CHIP interface control is used, only the first signal, a, is required.

Approaches, Trade-offs, and Solution.

Approaches. As was previously indicated, there are basically two approaches to implementing the chip interface controls. One approach is to place all the necessary chip interface and pipeline control circuitry on each WFTA chip, and have each chip control a particular memory in the pipeline. Predefined communications protocols would be used to coordinate transfers between chips, and to signal neighboring processors when a memory bank was free or being used. The other approach is to place most of this control circuitry off-chip on a special Interface Chip. This Interface Chip would be able to coordinate two or more processors actions for them by arbitrating the memory and controlling a small subset of control lines that would still run to each WFTA chip.

Trade-offs. The biggest advantage to the on-chip interface control is that the WFTA chip would then be self-contained. Only WFTA chips and memory chips would be needed to implement the PFA pipelined architecture. Most interfaces would be strictly asynchronous, and the overhead would be brought to its lowest level. Only active and W/D processors would have to run on the same clock signals.

The advantages of an off-chip controller, such as a WFTA Interface Chip, are many. First, fewer number of WFTA chip pins are needed to implement the interface control when it resides off-chip. The W/D startup signal and four of the pipelined memory access signals are not needed in the off-chip configuration. This accounts for a savings of five pins. The W/D startup signal is not needed since the Interface Chip can coordinate the synchronization of the watchdog and active chips by sending them the OPR signal at the same time. The Interface Chip would be able to control memories in a pipelined architecture by knowing when each

WFTA chip finished reading in its data and finished outputting its results. The Interface Chip would be able to control all memories and switch them when required. Since it controlled the memories and started each chip, it would have little problem coordinating the WFTA chips.

The second advantage of the off-chip controller is its flexibility. If the chip interface circuitry is placed on-chip, it may limit the possible configuration of WFTA architectures. The on-chip solution may only be able to handle the pipelined architecture with all three WFTA chips in line. An Interface Chip, however, could be instructed by the host cn the current chip configuration, and control the chips' operations in a manner appropriate for that architecture, be it pipelined, shared memory, or a single WFTA chip.

The third advantage of the off-chip controller is its simplicity over the on-chip alternative. The off-chip controller can be much simpler since it controls the memories and the WFTA chips, and all communications pass through it. The complicated hand-shaking circuitry required for the on-chip configuration does not have to be built if the off-chip controller is used. The chip can then employ simple operate and done signal lines. Implementing the W/D mode of operation also becomes a much simpler task if an off-chip controller can be responsible for synchronizing the active and watchdog WFTA chips rather than the chips attempting to synchronize themselves.

The last, and perhaps the most important, advantage of the off-chip controller is the added fault-tolerance it provides. In the on-chip interface control configuration, a watchdog chip must have its OPR line hardwired from the active chip's W/D startup pin. In the off-chip interface control configuration, all chip OPR lines (as well as all W/D mode lines) are wired to the WFTA Interface Chip. Thus, the Interface Chip (or the host processor through the Interface Chip) can dynamically decide which chips are the active and which chips are the watchdog. In the on-chip configuration, active and watchdog WFTA chips may be switched only if the wiring is redone. The off-chip controller could also poll three or more WFTA chips to determine which of the three is really bad if some watchdog discrepancy occurs. The bad WFTA chip could be inactivated by the controller. Thus, the off-chip controller or Interface chip allows triple-cyclic redundancy checking without hardware reconfiguration. Figure II-7 illustrates the general configuration of a WFTA processing element in the pipelined PFA.

Solution. The advantages of the Interface Chip over on-chip interface control are many, and therefore, the Interface Chip will be used to control the WFTA chips using the control signals described in the previous section.

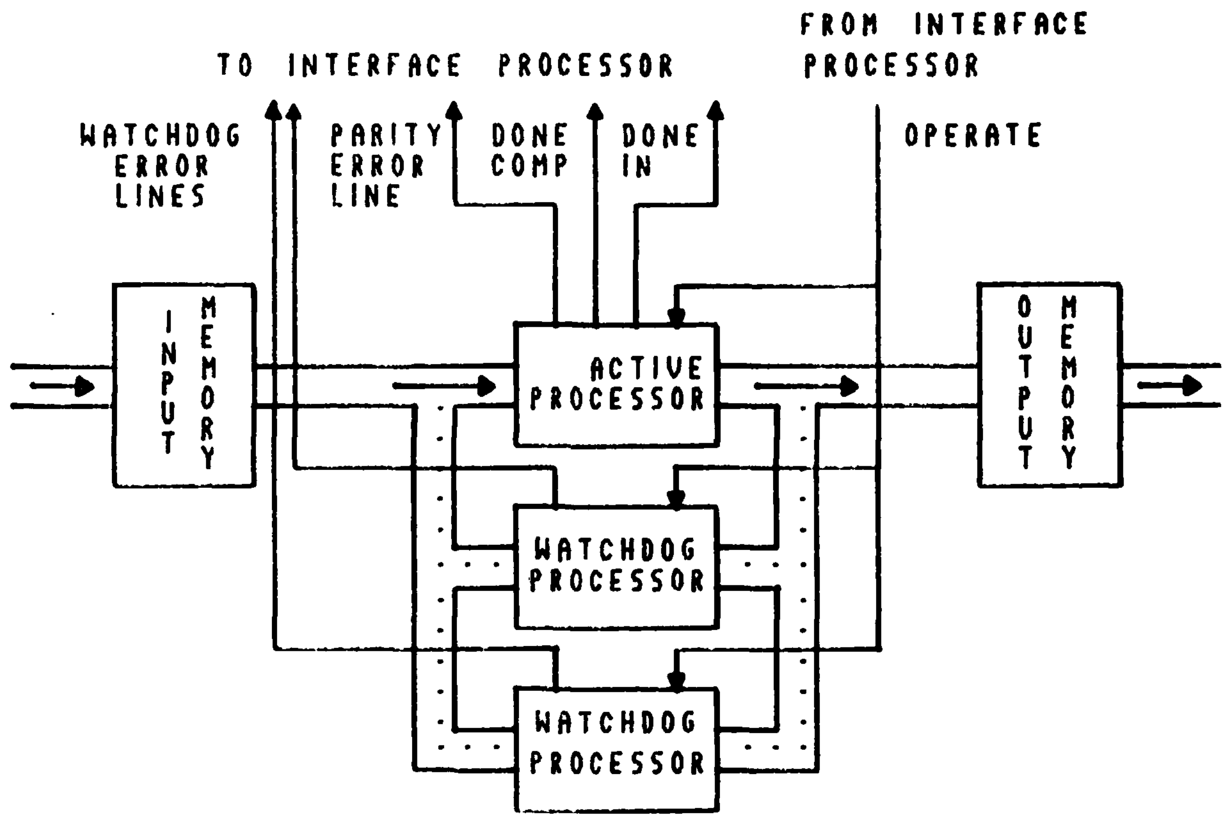


Figure II-7 WFTA Processing Element

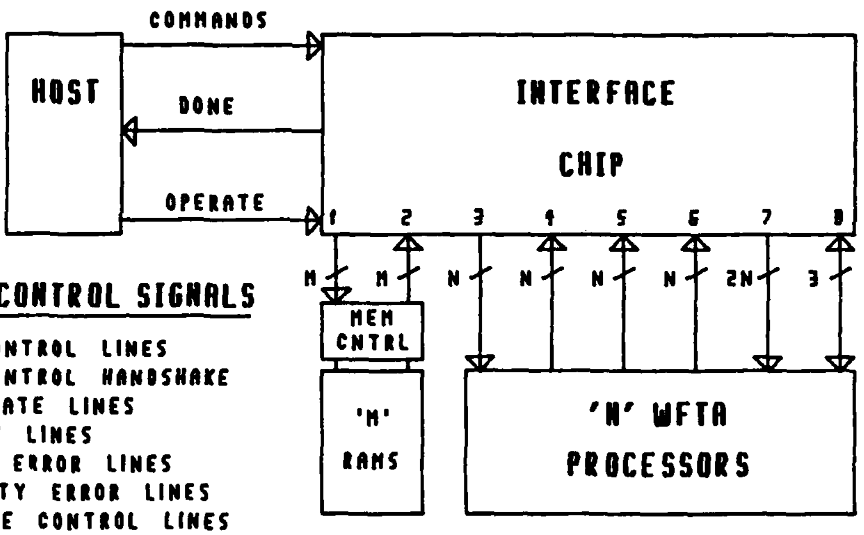
Configuration for the WFTA Interface Chip. The proposed configuration for the WFTA Interface Chip is shown in Figure II-8.

The Interface Chip will receive commands from the host processor, and control the WFTA chips and memory banks in accordance with those commands. The WFTA Interface Chip will control the W/D modes of operation, OPR signal timing to the chips, state loading such as scale factors and DFT size, scale factor passing from one WFTA to the next, memories, and the like.

The memory controller module is a large crossbar switch that switches the memory banks between two WFTA processors or a WFTA processor and the host. After a WFTA processor outputs all transform data to a memory bank, the next processor in the pipeline is connected to that memory bank, in order to read out the data. This technique allows all processors in a pipeline to operate simultaneously, once the pipeline is filled. The memory controller switches two sets of data, address, and write strobe lines that are routed directly from two or more processors. Each memory bank in a pipeline is almost always connected to one processor or the other through the memory controller switch.

The Interface Chip's flexibility and potential for fault detection, isolation, and correction has already been presented. However, Figures II-9 and II-10 present two possible WFTA configurations for two different DFT applications.

Since the WFTA Interface Chip was not designed pursuant to this thesis effort, it will not be presented in the Design Chapter (Chapter III). It was presented in this chapter, however, to show the viability of the WFTA chip's control design.



INTERFACE CONTROL SIGNALS

1 MEMORY CONTROL LINES
2 MEMORY CONTROL HANDSHAKE
3 WFTA OPERATE LINES
4 WFTA DONE LINES
5 WFTA W/D ERROR LINES
6 WFTA PARITY ERROR LINES
7 LOAD STATE CONTROL LINES
8 SCALE/DFT SIZE-MODE BUS

Figure II-8 WFTA Interface Chip.

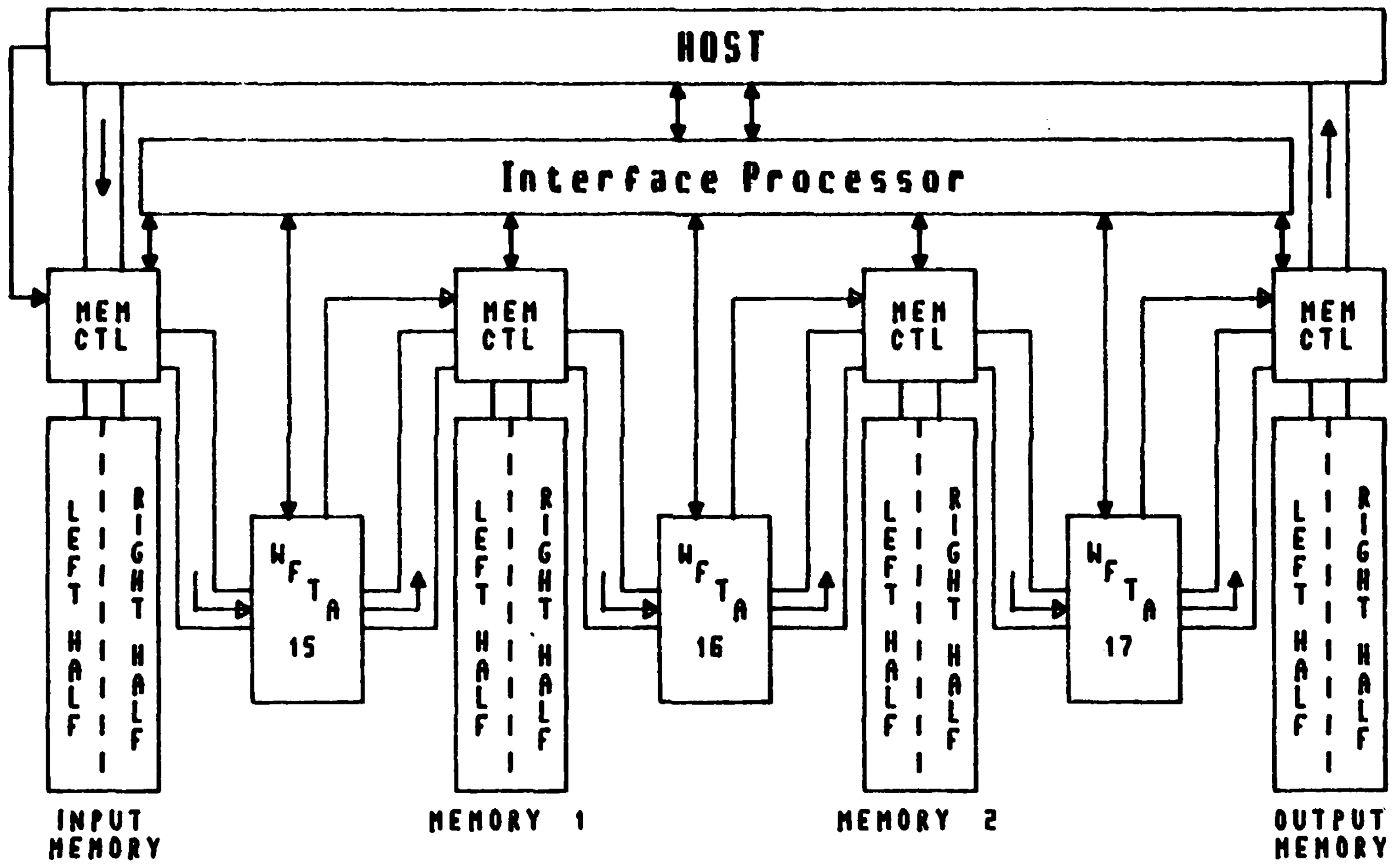


Figure II-9 Pipelined DFT Architecture

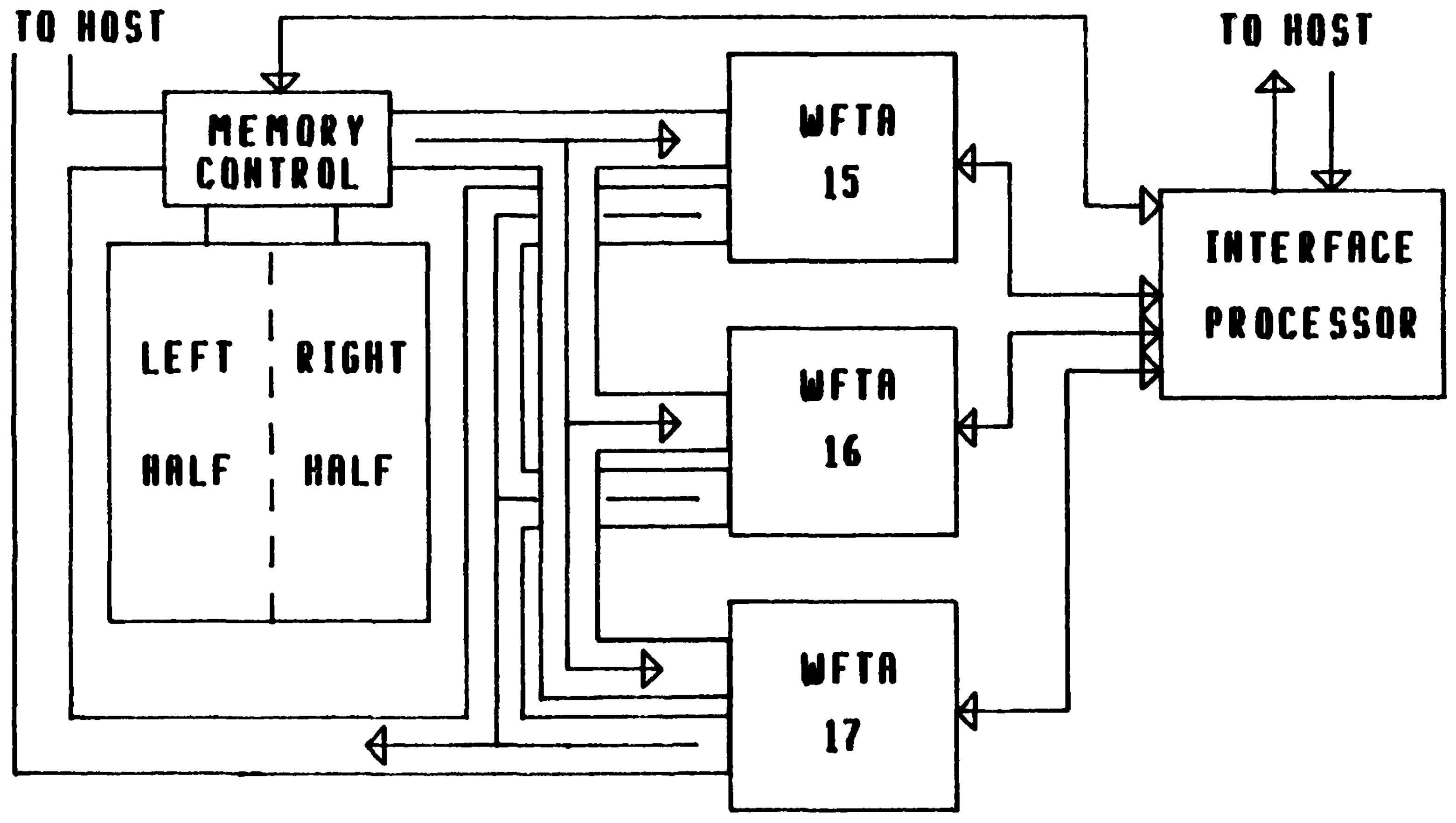


Figure II-10 Shared Memory DFT Architecture

III. Design

Method

Overview. The design of the WFTA control modules was part of the larger task of designing the entire WFTA chip. Major subsections of the control modules were designed, and these cells were used as the building blocks of the control circuitry.

Since no previous design projects at AFIT were implemented using CMOS technology, all arithmetic and control section cells had to be designed from scratch. Much time and effort was taken to design the critical cells to minimize the area used, insure functionality, and maximize speed.

Numerous test chips were designed and fabricated to test the functionality and speed of the modules designed. A test chip was fabricated for the Control Sequencer circuit and for the XROM address generation circuit. The results of these test chips are discussed in Chapter V. The WFTA chip was designed using the tested control and arithmetic module designs.

Tools. A number of Computer-Aided Design (CAD) tools were used in designing the cells, modules, and integrated circuits required for the WFTA chips.

All the circuits were created and edited using Caesar (Ousterhout, 1983). Caesar is an interactive VLSI design tool that uses a color graphics display terminal and a digitizing pad to aid the circuit designer. Design rule checking was performed on the Caesar circuits using Lyra (Arnold, 1983).

Simulation tools were limited to SPICE (Nagel and others, 1983) and custom made simulation/validation efforts (Collins, 1985). The SPICE simulation program was used primarily to determine if the worst-case paths of the design operated at the required speed. Visual inspection, test chips, and the custom made simulations were the primary functionality checkers.

Two final tools were invaluable in helping to verify an integrated circuit ready for submission. These two tools were Mextra (Fitzpatrick, 1983) and C Stat, a CMOS version of Stat (Baker, 1985) developed at AFIT. These tools read an integrated circuit layout description in CIF and provide information on aliased or unconnected nodes and transistors which could not be affected by the inputs or affect the outputs.

Design Rules. The circuits were designed using a scalable design rule set. The design rules facilitate designing for the 1.2 micron technology and processing at either 1.2 or 3 microns. The CMOS design rules used are listed in Appendix A.

Control Sequencer

Operation. The Control Sequencer design is depicted in Figure III-1. The Control Sequencer dimensions are 1000

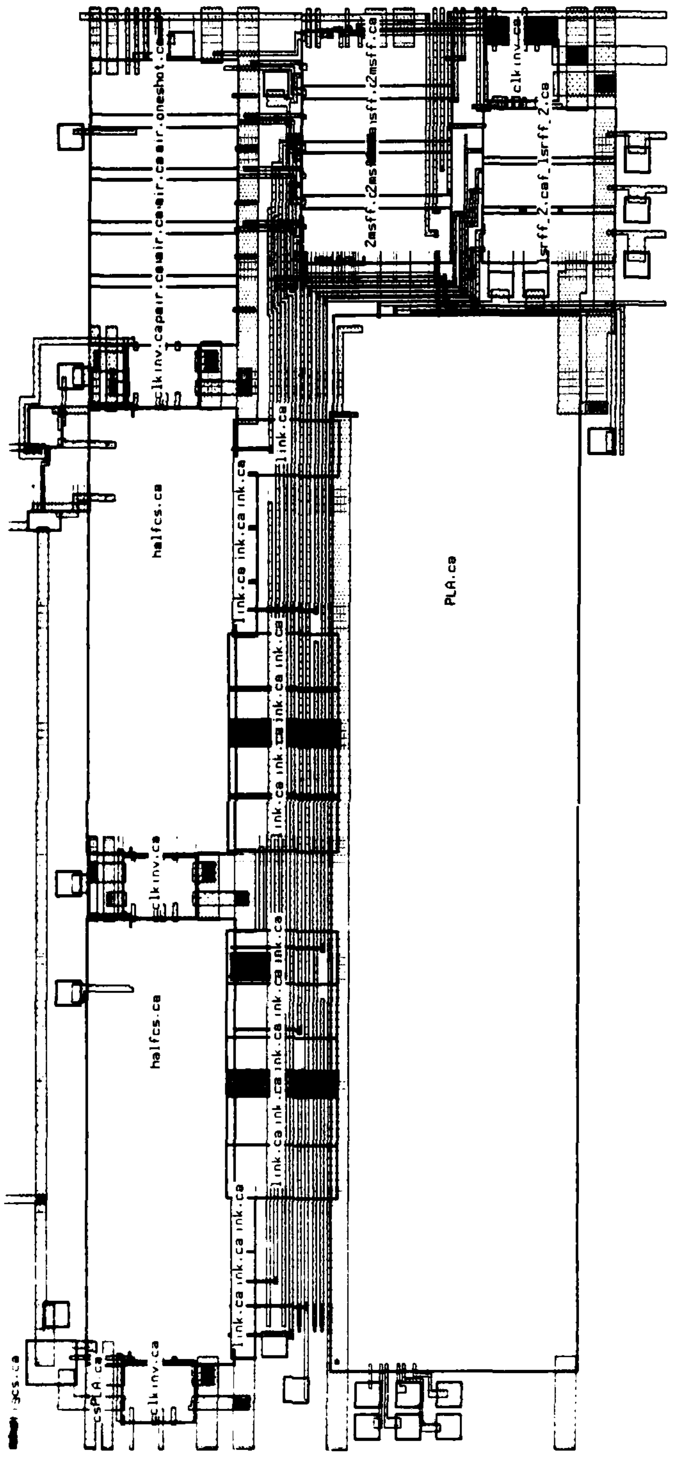


Figure III-1 Control Sequencer

lambda by 2400 lambda and fit well within the allotted chip area (see Figure I-10). The Control Sequencer receives a level signal, OPR, from off chip to start operation. The circuit's one-shot produces a single pulse from the rising OPR signal, and this pulse will be passed down the MSFF chain which acts as a ring counter. The initialization column outputs signals to various modules on the chip (primarily the XROM address generating circuit) to set up the circuitry for the DFT calculation cycles. The MSFF chain continuously cycles the single bit of the ring counter through all bit positions and back to the first position as long as the DONE signal is not active. As each MSFF receives the passed bit, its output not only goes to the next MSFF in the chain, but also to the PLA product terms that correspond to the present clock cycle of the timing diagram. When a product term is driven high by the ring counter it may affect the outputs of the OR-plane if the AND-plane and OR-plane conditions are met. The output of the ring counter MSFF may bypass the PLA and be input directly to an output MSFF or SRFF if the signal is always required (regardless of the external PLA inputs), and if the clock cycle signal is not OR'ed with any other clock cycle signal.

The control signals of the timing diagram are output to the XROM and arithmetic circuitry from the output MSFFs and SRFFs. These flip-flops' outputs are controlled by the sequence of pulses received from the Control Sequencer's PLA output signals or the ring counter output pulses. The Control Sequencer continues to output the same pattern of control signals as the bit is cycled around the MSFF ring counter. This process continues until the DONE signal (generated by the ROM counter/subtractor circuit) is raised and the bit is not allowed to cycle back to the first MSFF in the ring counter.

Testability Considerations. VLSI circuits are inherently difficult to test, and it is even harder to isolate faults once they are detected. There are at least three basic reasons why VLSI circuits are so difficult to test. First, the number of possible faults is extremely large. A VLSI circuit contains thousands of transistors and interconnect lines, all individually subject to failure. Second, access to all the internal transistor outputs and interconnect lines is severely limited by the small number of I/O ports available. Third, the large number of faults that can occur will require numerous test vectors to determine correctness or fault (Hayes and McCluskey, 1980:17).

Because of the difficulty in testing VLSI circuits, techniques to improve the testability of the WFTA chip had to be incorporated into the design sequence. Three basic techniques were used to help improve the test controllability and observability of the WFTA chip.

The first technique was to serially chain all control output flip-flops and ring counter flip-flops to external pins (see Figure III-2). The serial connections can be controlled by a pin input called TEST. If TEST is high, the test configuration is enabled and groups of output flip-flops become serially linked through transmission gates. The chip's pins take on a test configuration also, and each group of linked cells are tied to an input and an output pad. The contents of the output flip-flops may be clocked out to the output pin in order to observe the state of the flip-flops, or test vectors may be loaded into the flip-flops from the input pins. This approach aids in fault isolation in both the PLA and arithmetic circuitry, and allows testing of the arithmetic circuitry to continue if the control circuitry fails.

The second technique was to connect the arithmetic and control circuitry to two separate clock pins. This approach allows testing of the arithmetic circuitry to be performed for each control signal vector output from the control circuitry. The state of the arithmetic circuitry can be left as is while the next correct Control Sequencer and XROM output vectors are clocked into the output MSFFs through the test ports. Then the arithmetic circuitry can be clocked to perform the required multiplications and additions. In this manner, the correct operation of the WFTA chip could be realized if the control circuitry were not functioning properly.

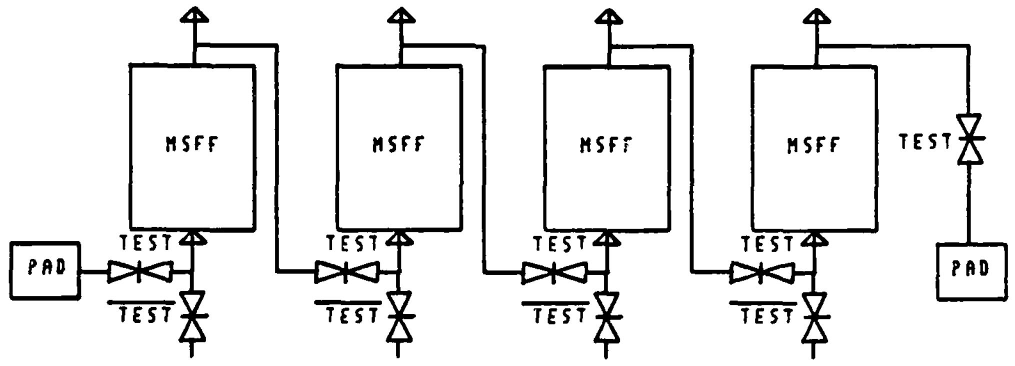


Figure III-2 Test Chaining.

The last technique employed was to provide numerous probe points on all chips fabricated. This approach increased the observability and sometimes controllability of circuit nodes beyond those that were connected to the chip's package pins.

These three techniques were integrated into the design of the WFTA. Groups of cells that are modified to function in the test configuration when TEST is high are the output flip-flops and the chain of MSFFs in the ring counter.

Control Sequencer Cells. The cells that make up the Control Sequencer (CS) can be divided into three main groups: the MSFF chain, the PLA, and the output MSFFs and SRFFs. Each of these groups and selected cells that are used in each group will be presented below. Descriptions and schematic diagrams of the cell circuits will be given. CIF-plots of certain cell features are also presented.

MSFF Chain. The MSFF chain is made up of the initialization column and the ring counter. Each of these sections is mostly composed of a chain of double MSFF cells called "pair." Pair is two resettable static delay master/slave flip-flops whose devices have been sized for speed and whose layout area has been compressed. Pair consists of two MSFFs because each MSFF shares common clock and reset signals. Each MSFF output has two output drivers, and a first metal path to the input of the next MSFF in the chain. The two output drivers are used to output signals directly to output flip-flops and/or through the PLA for each clock cycle of the ring counter. Depending on the timing diagram, both paths may be used, both outputs may be tied to the PLA, or neither may be used at all. A schematic diagram of one of the MSFFs in pair is shown in Figure III-3.

The cell size of pair is 76 lambda by 200 lambda. Pair has horizontal Vdd and GND lines running in first metal while all clock and reset lines run vertically in second metal.

A one-shot cell is at the top of the initialization column. The one-shot is made from a pair cell with

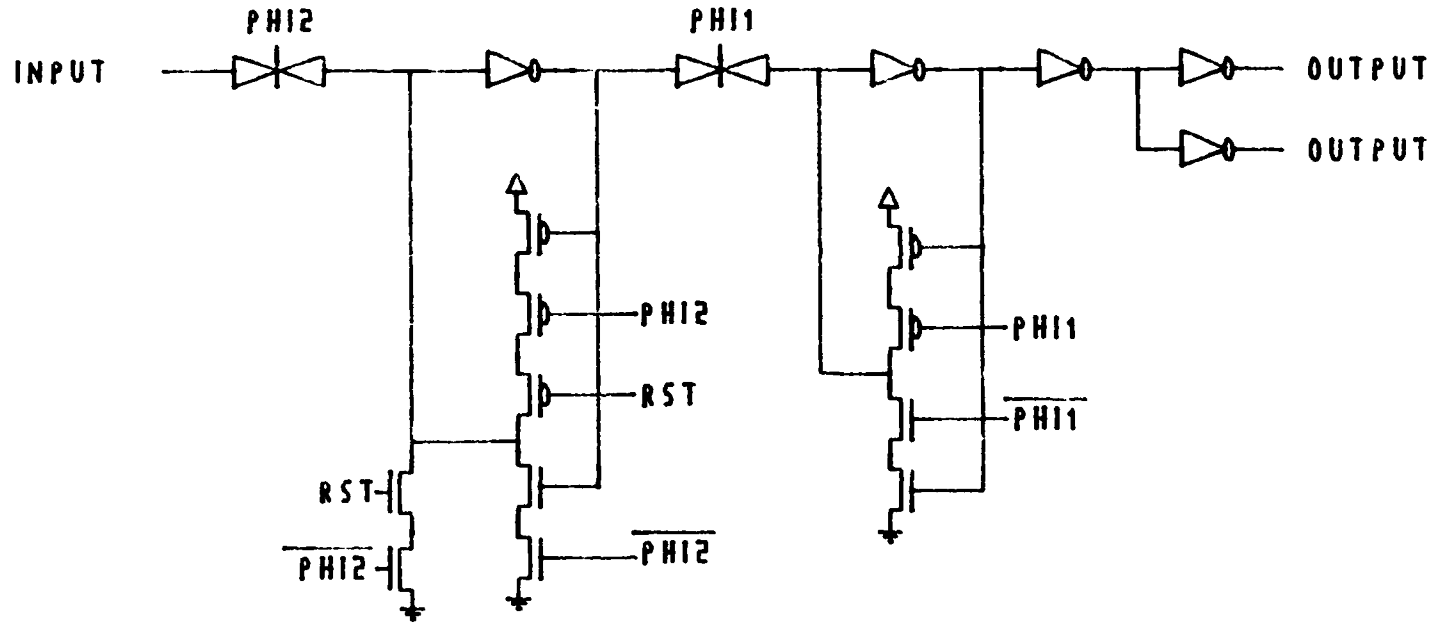


Figure III-3 Master/Slave Flip-Flop

its output drivers replaced with a one-shot circuit. This circuit takes the output of the first MSFF and ANDs it with the inverted output of the second MSFF. A schematic of the one-shot is shown in Figure III-4.

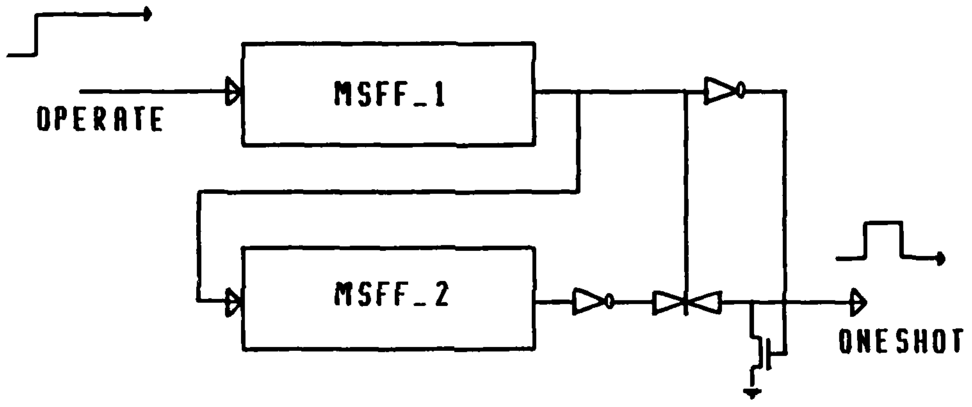


Figure III-4 One-shot

Thus, if the level OPR signal is input to the first MSFF the output of the one-shot will be a pulse that is high only during the one clock cycle that the first MSFF outputs a one while the second MSFF still outputs a zero.

The last element of the group is the loop-back circuitry for the ring counter. Each time the ring counter bit reaches the last output, it must be fed back to the input of the first MSFF in the ring counter (the MSFF immediately below the last MSFF in the initialization column). Since two inputs must feed into the first MSFF of the ring counter, an OR-gate must be used. Additionally, when the DONE signal goes high the bit must be prevented from looping back. Thus, an AND-gate must also be placed on the loop-back path. This loop-back circuitry is shown in Figure III-5.

CS PLA. The CS PLA is made up of the AND input drivers, the AND-plane, the OR-plane, and the OR output drivers. The CS PLA is a NOR-based PLA that seeks to improve its device current drive to loading capacitances ratio by using a "donut" device in both the AND and OR planes. The donut device cell is shown in Figure III-6. The high current drive capability of the donut device is used to rapidly pull down the product term or OR-column output line of the PLA. The high current drive is combined with the small drain and sidewall capacitances of the donut devices that are connected to these same two lines to provide for a very fast PLA. The donut device's area is minimal for the large gate width of the device allowing for a fairly dense array structure. It should be noted here that the drive capability, small area, and small drain capacitance of the donut structure is desirable in many

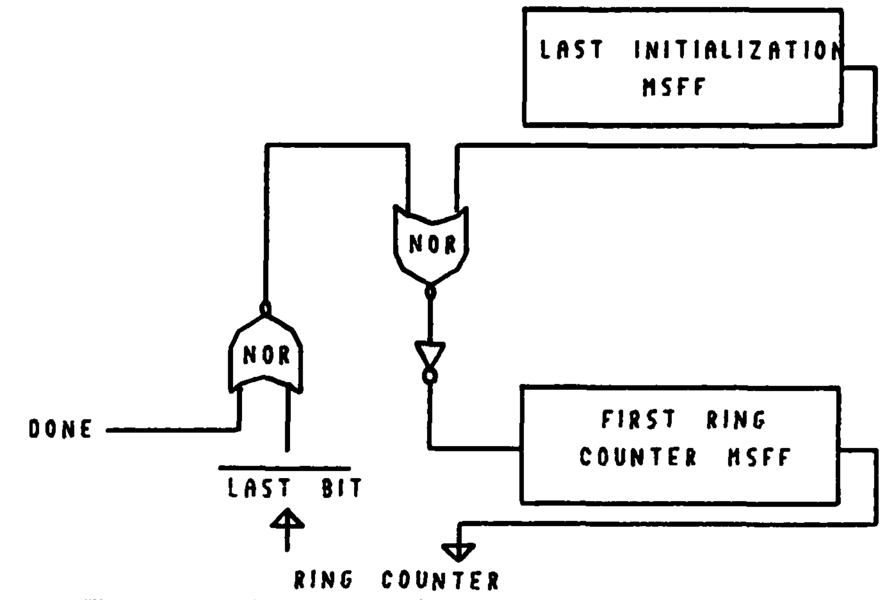


Figure III-5 Ring Counter Loop-back Circuitry

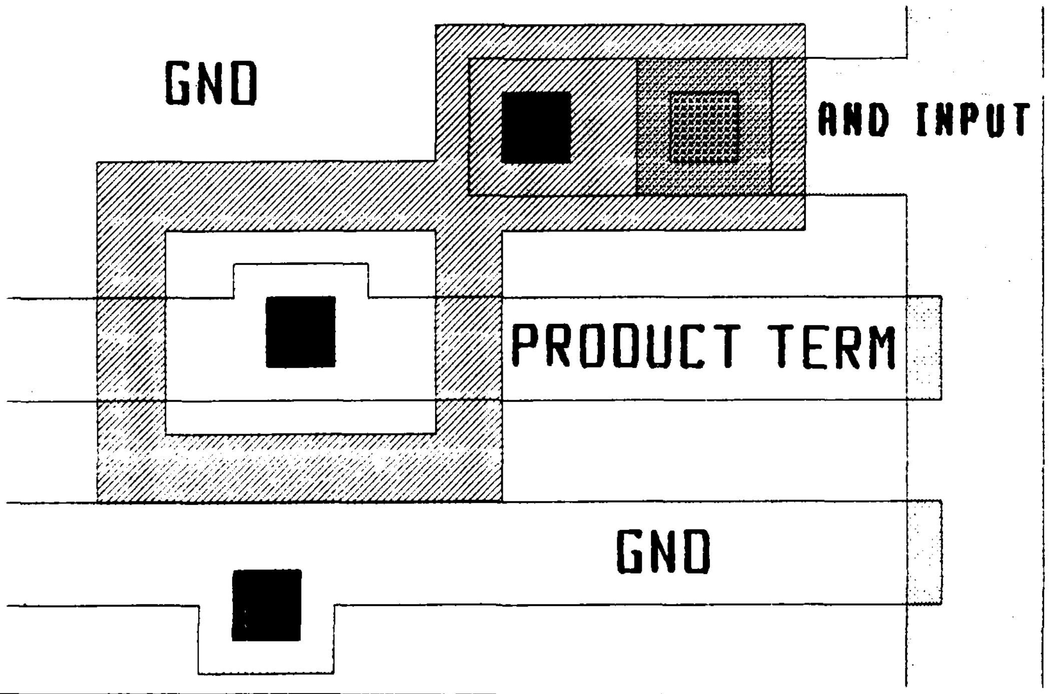


Figure III-6 PLA Donut Device

situations (not just a PLA). Thus, the donut structure is used throughout the control circuitry especially when large current drive is needed in a minimum amount of area.

Each ring counter output has two product term inputs available (one for each of the ring counter MSFF's two output drivers). The two product term lines for one clock cycle state are used for instances when the timing diagram requires more than one AND term combination of a present state and external inputs. Since there are two PLA cells for each MSFF, the cells are 19 lambda tall. The AND-plane cells are 25 lambda wide and the OR-plane cells are 24 lambda wide. The product term and GND lines run horizontally in the array and the AND-plane input columns and OR-plane output columns run vertically. The schematic for the PLA is shown in Figure III-7.

The input drivers and the output drivers are made up of donut device inverters for large gate width and correspondingly large current drive. Static pull-ups are placed on the OR-column outputs while the ring counter drivers serve as the pull-up on the active product term line.

The n-device pulldowns in the array of Figure III-7 are the donut devices. The array is personalized using contacts and second metal strips. The contacts connect drains to product term lines in the AND-plane and gates to product term lines in the OR-plane. The second metal

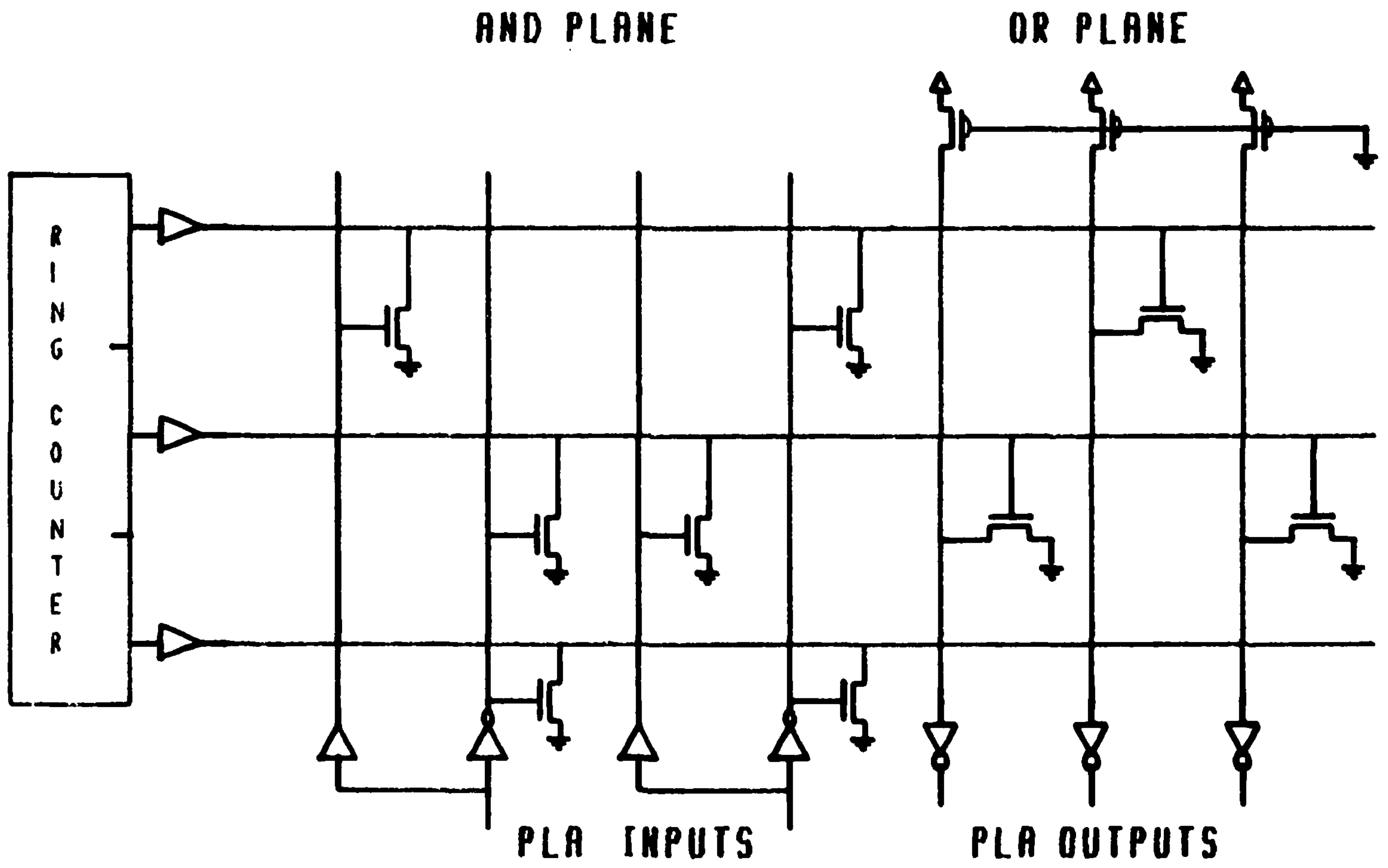


Figure III-7 PLA Schematic

strips connect the input columns to gates in the AND-plane and output columns to drains in the OR-plane. This approach allows a standard PLA chip to be fabricated up to the contact layer before a personalization must be applied.

The PLA's OR-plane employs a space saving technique of running two output column lines to two OR output drivers for every OR donut cell column. This technique allows a type of PLA folding where two OR-plane outputs that have mutually exclusive product term inputs can be obtained from one OR-plane column of devices.

MSFFS and SRFFS. These output flip-flops are resettable and have large staged-up, donut output drivers. The MSFF is made from the pair cell without the feedback loop and with the larger output drivers. In the output MSFF array, each flip-flop receives its own pulsed input and outputs that control pulse to the chip one clock cycle later. The schematic of the MSFF cell is shown in Figure III-8 (The test cell configuration is shown).

The SRFF consists of two phi 2 latches, set-reset circuitry, and one phi 1 latch with a large, staged-up donut output driver. The set-reset circuitry was placed between the phi 2 and phi 1 latches in order to minimize the delay to the phi 2 latch from the ring counter and PLA. The SRFF is depicted in Figure III-9 (the test cell configuration is shown). The cell sizes for the output flip-flops are 84 lambda by 250 lambda for a MSFF cell

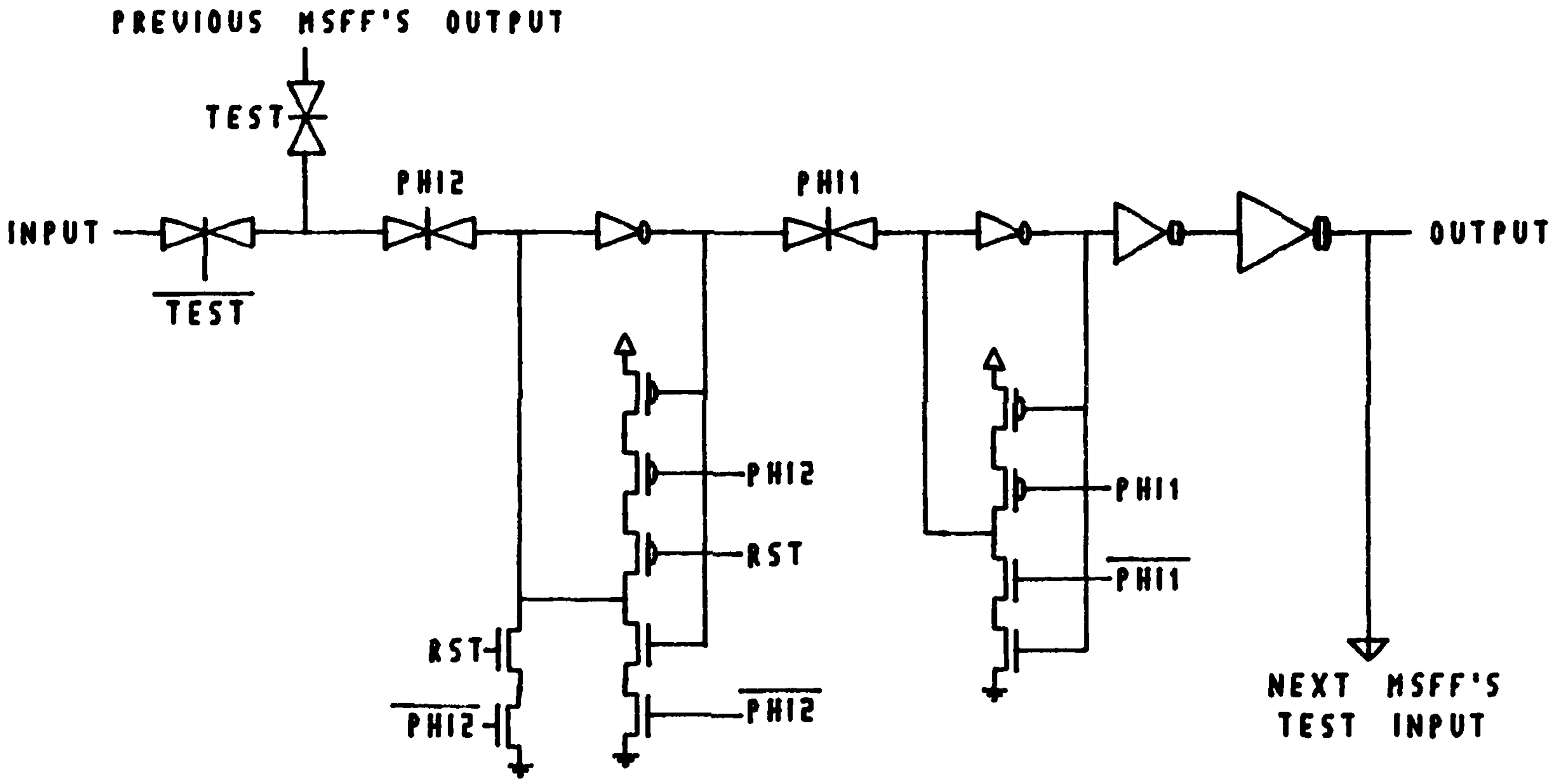


Figure III-B Output MSFF

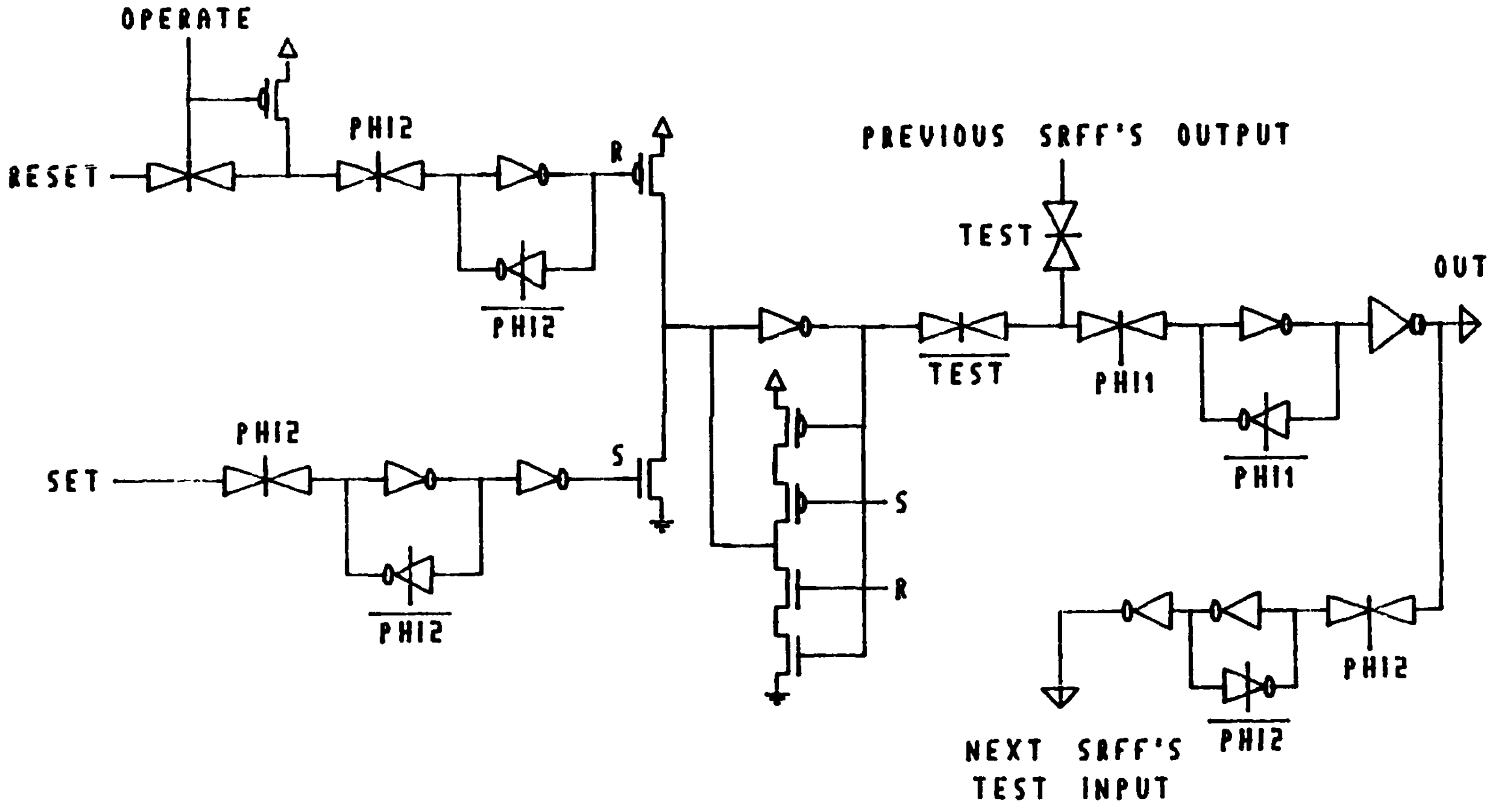


Figure III-9 Set/Reset Flip-Flop

which contains two MSFFs and their drivers) and 90 lambda by 225 lambda for a SRFF cell. The GND, Vdd, and clock lines run in the same manner as the pair cell.

SPICE Simulations. As was indicated in the method section of this chapter, the SPICE program was used to simulated the worst-case path of the control circuitry. The worst-case path in the Control Sequencer is one that travels from the MSFF in the ring counter, to the PLA, through the product term and output line that has the worst-case personalization, out to a SRFF cell, and finally out to the chip. A schematic of this worst-case path is shown in Figure III-10.

Since the signal is allowed two clock cycles to traverse this path, the SPICE results may be shown in two separate illustrations. In the first section, the signal must travel through the MSFF and the PLA and arrive at the phi 2 latch of the output SRFF in one clock cycle. The slowest portion of this section is the path from the output latch of the MSFF to the input latch of the output SRFF through the PLA. Figure III-11 shows the SPICE output for this path for a rising edge from the MSFF and Figure III-12 shows the SPICE output for the falling edge.

As the SPICE output shows, the maximum delay is on the order of 15 nS or a clocking frequency of over 60 MHz. These results are for the 3 micron process and parameters as listed in Appendix E. The speed of operation for the

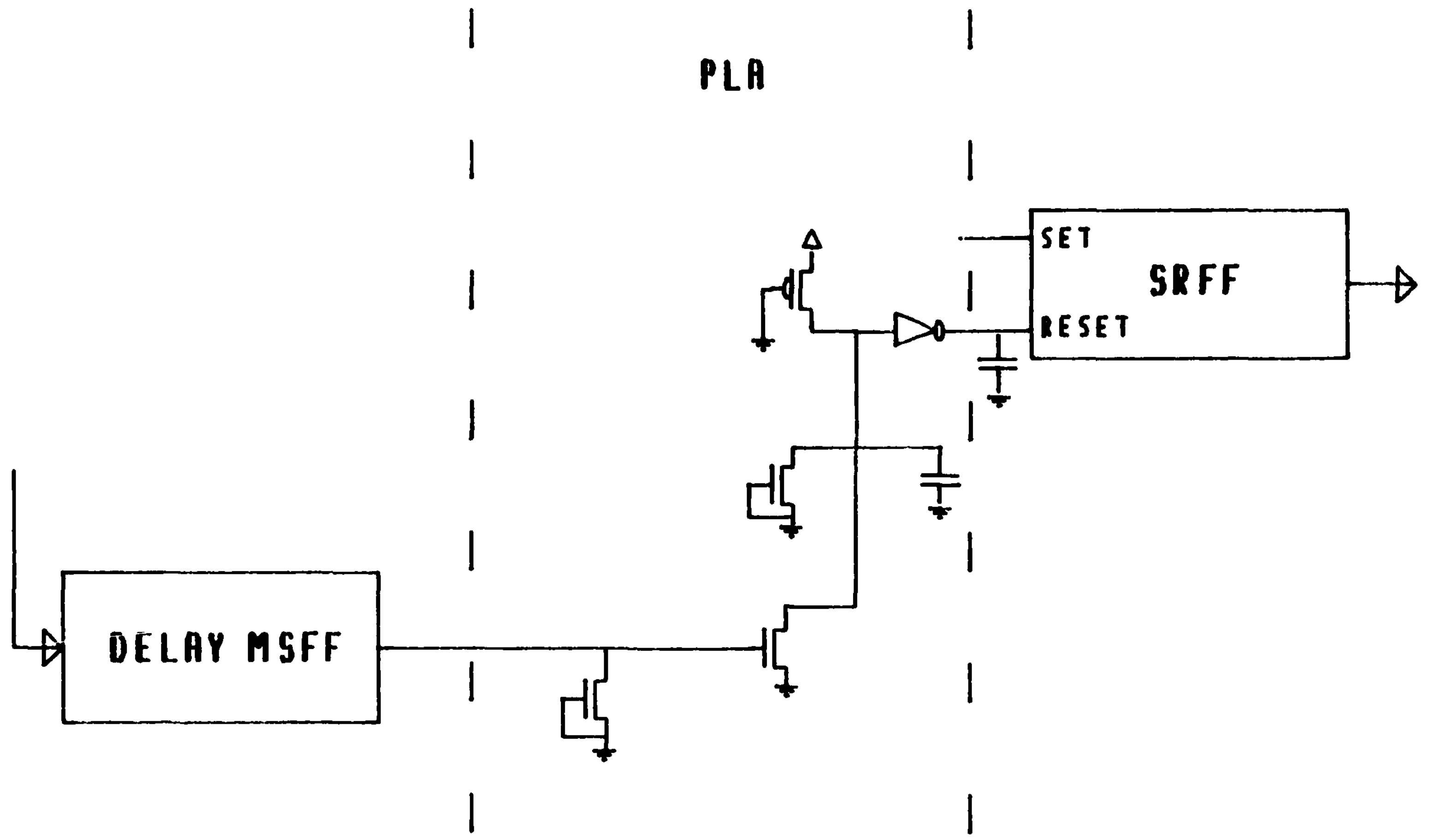


Figure III-10 Control Sequencer Worst-Case Path

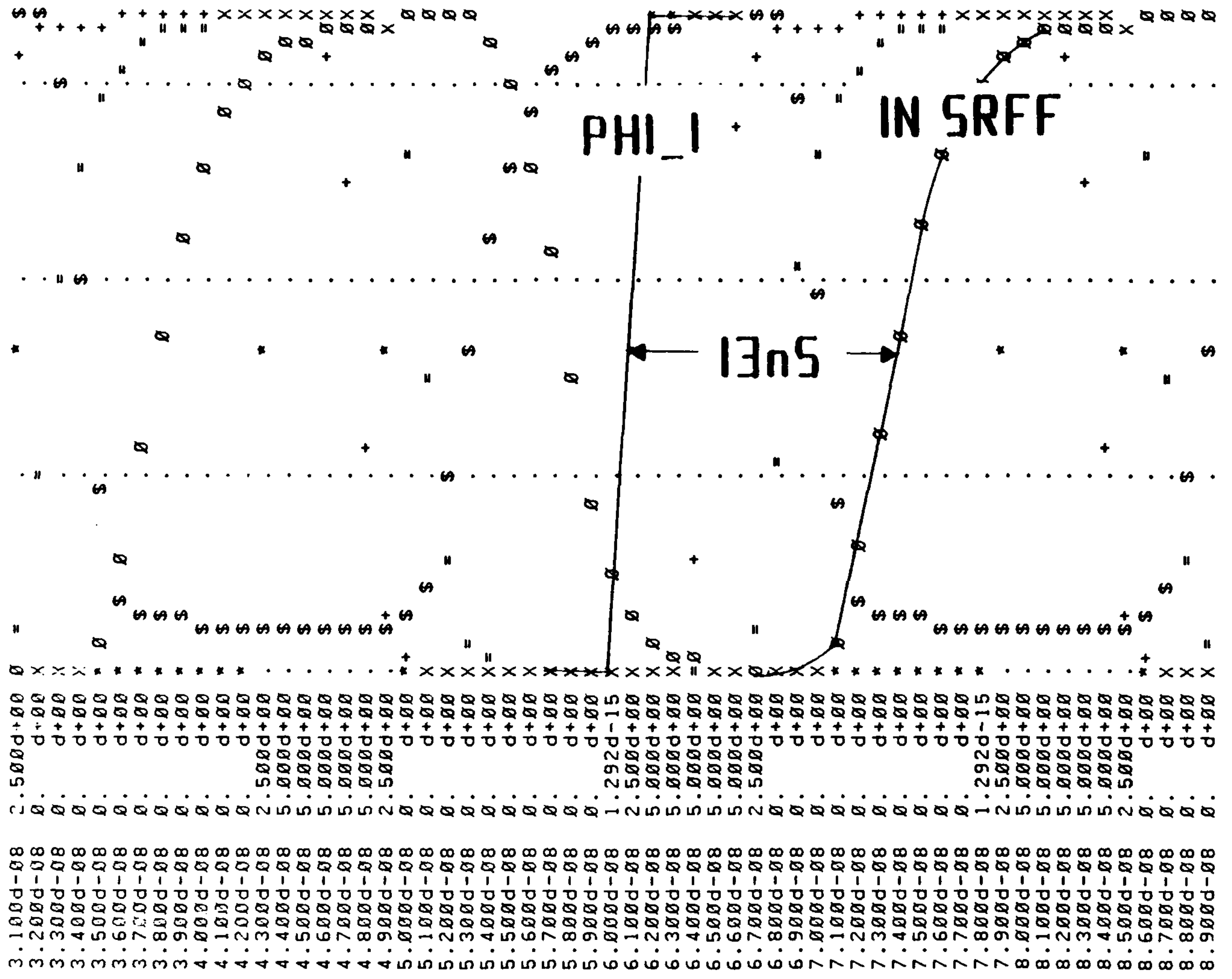


Figure III-11 SPICE: PLA - Rising Edge

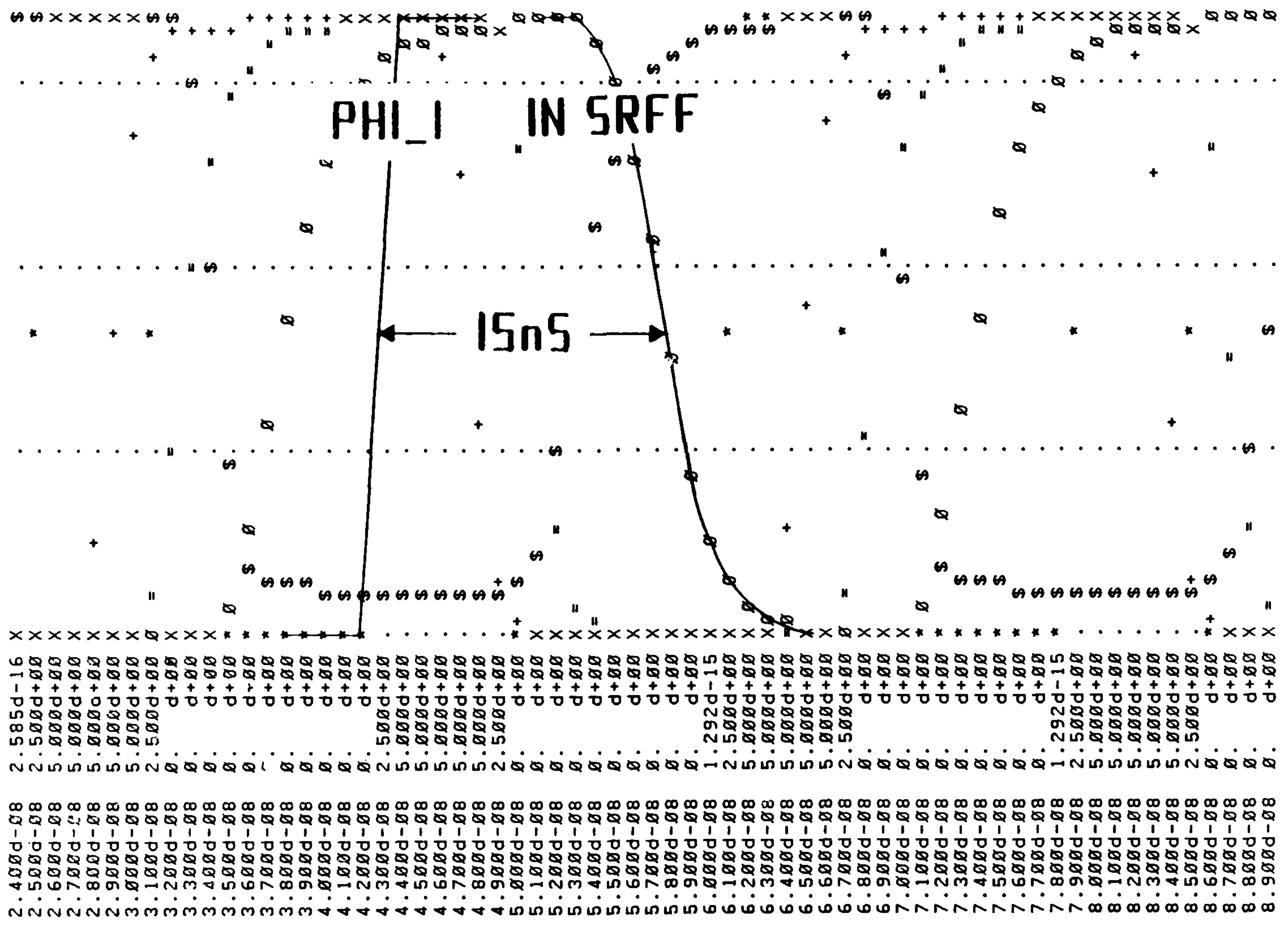
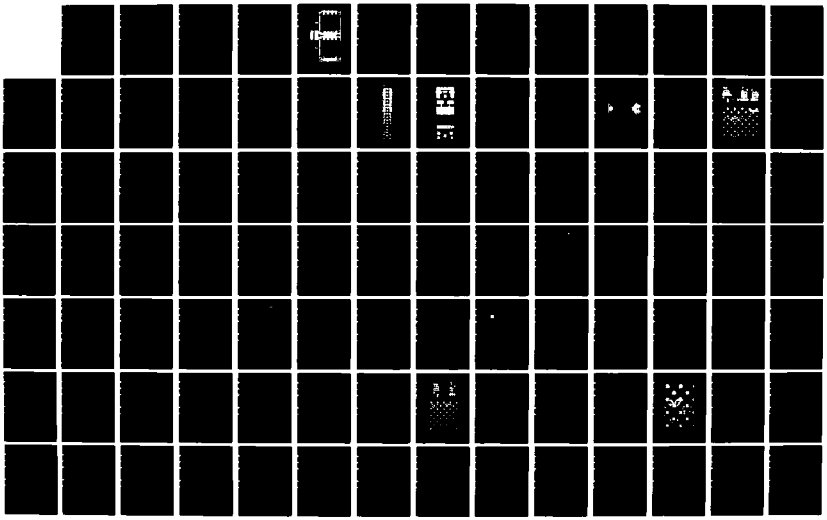
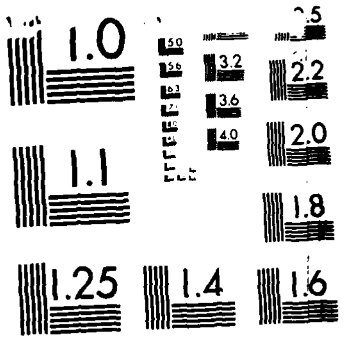


Figure III-12 SPICE: PLA - Falling Edge

AD-A164 043   CONTROL CIRCUITRY FOR HIGH SPEED VLSI (VERY LARGE SCALE   2/4
INTEGRATION) WINO . (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI   P C ROSSBACH
UNCLASSIFIED   DEC 85 AFIT/GE/ENG/85D-35   F/G 9/5   NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

circuit will improve for the 1.2 micron process due to the scaling effects (Mead and Conway, 1980:34).

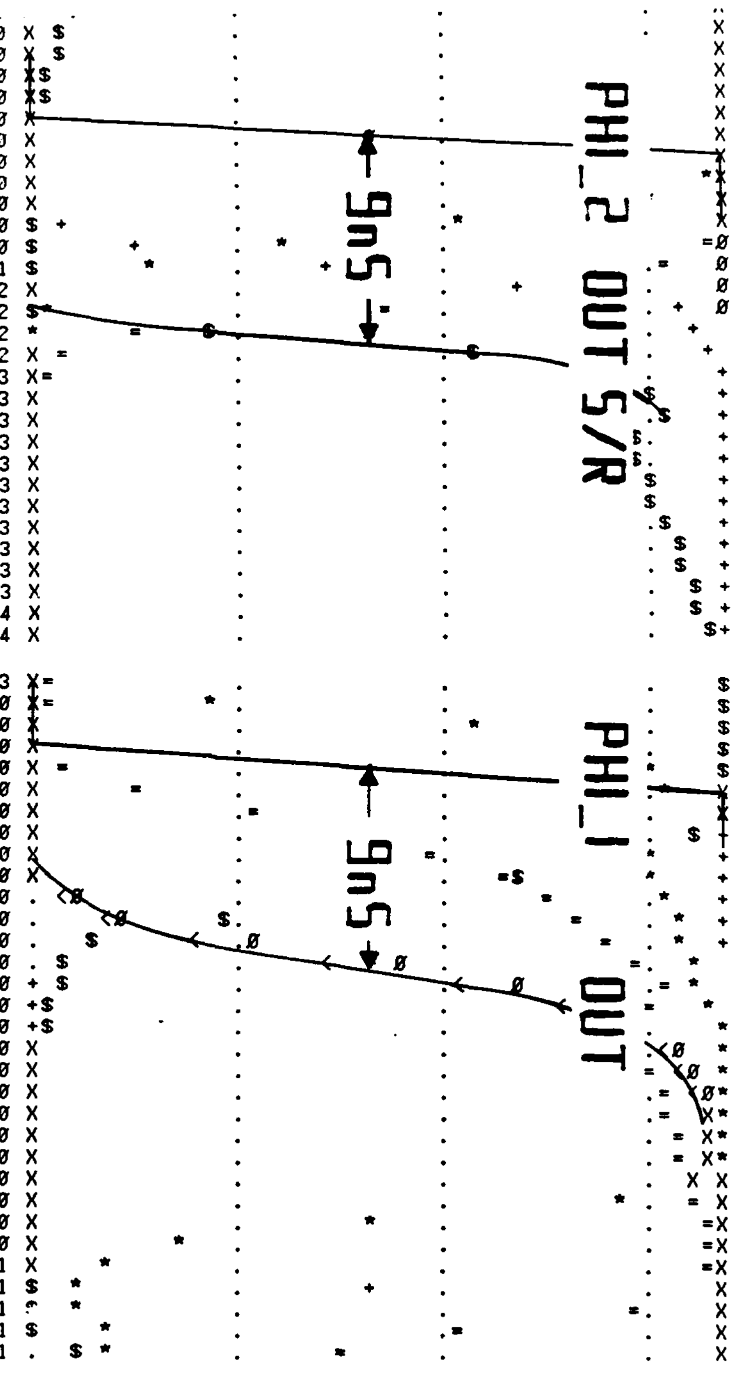
In the second section, the signal must travel through the SRFF cell and arrive at the phi 2 latches of arithmetic cells located on chip in one clock cycle. Figure III-13 shows the SPICE output for this path on a set control signal, and Figure III-14 for a reset control signal.

The SPICE output shows the maximum delay to be 18 nS or a clocking speed of over 55 MHz. Again these results will improve as the process is scaled down to 1.2 microns.

Thus, the SPICE simulations show that the Control Sequencer's design can achieve a speed of operation of over 55 MHz for the worst-case path in the 3 micron process. Operating speeds in the 1.2 micron process should reach over the 70 MHz target speed with little or no difficulty.

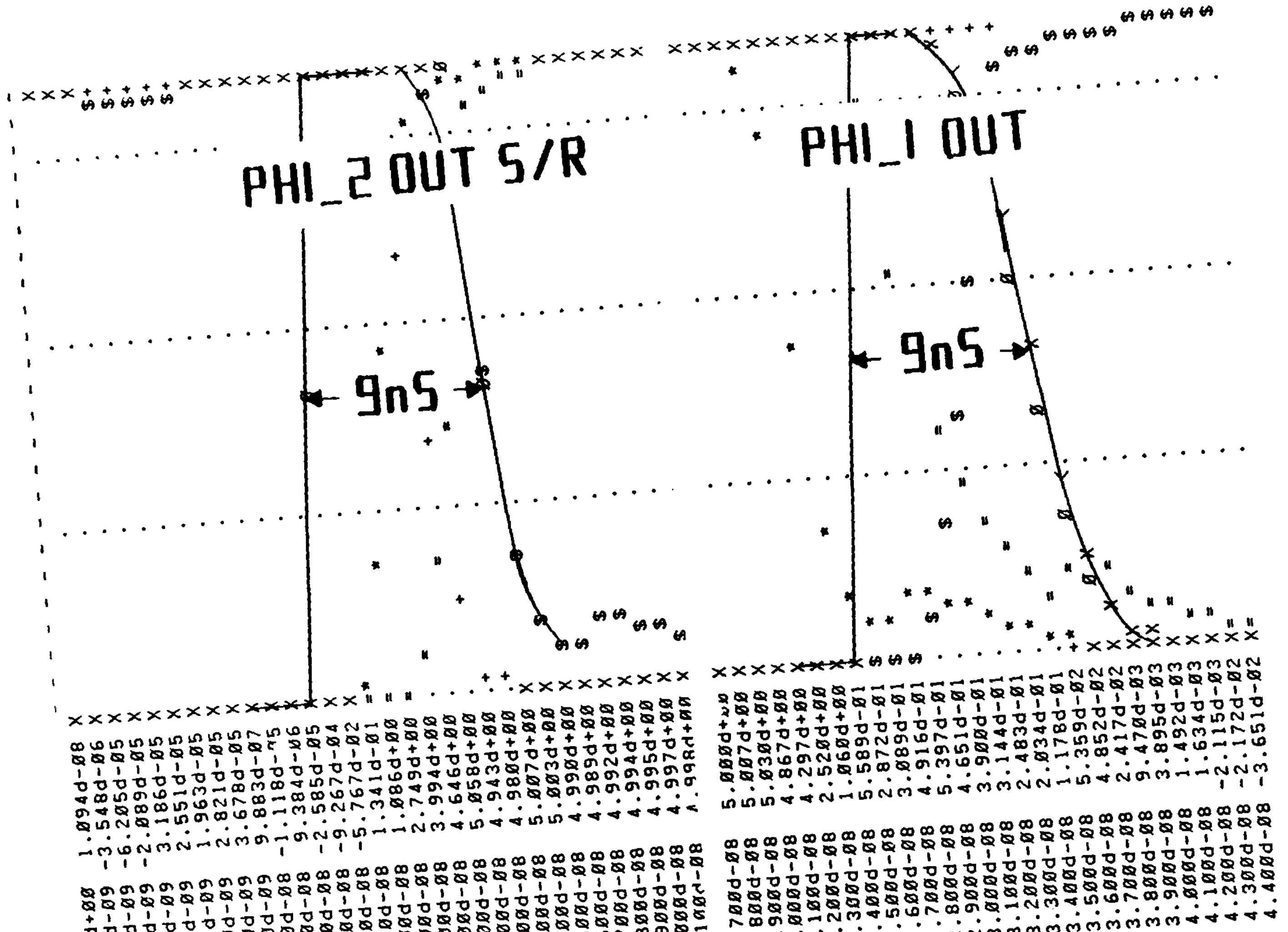
XROM Address Generator

XROM Operation. The XROM Address Generator design is depicted in Figure II-15. The XROM Address Generator dimensions are 2200 by 4000 lambda and fit within the allotted chip area (see Figure I-10). The XROM addresses are generated by the counter/subcontractor circuit and applied to the XROM's wordline PLA decoder (8), address column of the array (1), and 4 to 1 multiplex circuitry at the top of the XROM array (2). The addressing retrieves a single 48 bit word from the XROM and outputs it to the



```
3.200d-08   4.998d+00
3.300d-08   4.999d+00
3.400d-08   4.999d+00
3.500d-08   4.999d+00
3.600d-08   4.999d+00
3.700d-08   5.020d+00
3.800d-08   5.060d+00
3.900d-08   4.972d+00
4.000d-08   4.479d+00
4.100d-08   3.076d+00
4.200d-08   1.771d+00
4.300d-08   8.617d-01
4.400d-08   4.235d-02
4.500d-08   5.740d-02
4.600d-08   1.404d-02
4.700d-08   1.434d-02
4.800d-08  -6.757d-03
4.900d-08  -3.454d-03
5.000d-08   3.486d-03
5.100d-08   5.194d-03
5.200d-08   4.917d-03
5.300d-08   3.952d-03
5.400d-08   3.033d-03
5.500d-08   2.381d-03
5.600d-08   1.741d-03
5.700d-08   1.317d-03
5.800d-08   1.024d-03
5.900d-08   8.069d-04
6.000d-08   5.723d-04

4.500d-08  -6.419d-03
4.600d-08   1.235d+00
4.700d-08   3.167d+00
4.800d-08   4.119d+00
4.900d-08   4.494d+00
5.000d-08   4.569d+00
5.100d-08   4.414d+00
5.200d-08   4.397d+00
5.300d-08   4.474d+00
5.400d-08   4.550d+00
5.500d-08   4.624d+00
5.600d-08   4.697d+00
5.700d-08   4.747d+00
5.800d-08   4.790d+00
5.900d-08   4.873d+00
6.000d-08   4.981d+00
6.100d-08   4.994d+00
6.200d-08   4.997d+00
6.300d-08   4.999d+00
6.400d-08   4.999d+00
6.500d-08   4.998d+00
6.600d-08   5.007d+00
6.700d-08   5.033d+00
6.800d-08   4.874d+00
6.900d-08   4.298d+00
7.000d-08   2.517d+00
7.100d-08   1.059d+00
7.200d-08   5.585d-01
7.300d-08   2.870d-01
7.400d-08   3.071d-01
7.500d-08   4.835d-01
7.600d-08   5.314d-01
```

Figure III-13 SPICE: SRFF - Set



```
Ø.     d+ØØ    1.Ø94d-Ø8
1.ØØØd-Ø9     -3.548d-Ø6
2.ØØØd-Ø9     -6.2Ø5d-Ø5
3.ØØØd-Ø9     -2.Ø89d-Ø5
4.ØØØd-Ø9      3.186d-Ø5
5.ØØØd-Ø9      2.551d-Ø5
6.ØØØd-Ø9      1.963d-Ø5
7.ØØØd-Ø9      2.821d-Ø5
8.ØØØd-Ø9      3.678d-Ø5
9.ØØØd-Ø9      9.883d-Ø7
1.ØØØd-Ø8     -1.118d-Ø5
1.1ØØd-Ø8      9.384d-Ø6
1.2ØØd-Ø8     -2.585d-Ø5
1.3ØØd-Ø8     -9.267d-Ø4
1.4ØØd-Ø8     -5.767d-Ø2
1.5ØØd-Ø8      1.341d-Ø1
1.6ØØd-Ø8      1.Ø86d+ØØ
1.7ØØd-Ø8      2.749d+ØØ
1.8ØØd-Ø8      3.994d+ØØ
1.9ØØd-Ø8      4.646d+ØØ
2.ØØØd-Ø8      5.Ø58d+ØØ
2.1ØØd-Ø8      4.943d+ØØ
2.2ØØd-Ø8      4.98Ød+ØØ
2.3ØØd-Ø8      5.ØØ7d+ØØ
2.4ØØd-Ø8      5.ØØ3d+ØØ
2.5ØØd-Ø8      4.99Ød+ØØ
2.6ØØd-Ø8      4.989d+ØØ
2.7ØØd-Ø8      4.992d+ØØ
2.8ØØd-Ø8      4.994d+ØØ
2.9ØØd-Ø8      4.995d+ØØ
3.ØØØd-Ø8      4.997d+ØØ
3.1ØØd-Ø8      4.998d+ØØ

1.7ØØd-Ø8      5.ØØØd+ØØ
1.8ØØd-Ø8      5.ØØ7d+ØØ
1.9ØØd-Ø8      5.Ø3Ød+ØØ
2.ØØØd-Ø8      4.867d+ØØ
2.1ØØd-Ø8      4.297d+ØØ
2.2ØØd-Ø8      2.52Ød+ØØ
2.3ØØd-Ø8      1.Ø6Ød+ØØ
2.4ØØd-Ø8      5.589d-Ø1
2.5ØØd-Ø8      2.872d-Ø1
2.6ØØd-Ø8      3.Ø89d-Ø1
2.7ØØd-Ø8      4.916d-Ø1
2.8ØØd-Ø8      5.397d-Ø1
2.9ØØd-Ø8      4.651d-Ø1
3.ØØØd-Ø8      3.9ØØd-Ø1
3.1ØØd-Ø8      3.144d-Ø1
3.2ØØd-Ø8      2.483d-Ø1
3.3ØØd-Ø8      2.Ø34d-Ø1
3.4ØØd-Ø8      1.178d-Ø1
3.5ØØd-Ø8      5.359d-Ø2
3.6ØØd-Ø8      4.852d-Ø2
3.7ØØd-Ø8      2.417d-Ø2
3.8ØØd-Ø8      9.47Ød-Ø3
3.9ØØd-Ø8      3.895d-Ø3
4.ØØØd-Ø8      1.492d-Ø3
4.1ØØd-Ø8      1.634d-Ø3
4.2ØØd-Ø8     -2.115d-Ø3
4.3ØØd-Ø8     -2.172d-Ø2
4.4ØØd-Ø8     -3.651d-Ø2
```

Figure III-14 SPICE: SRFF - Reset

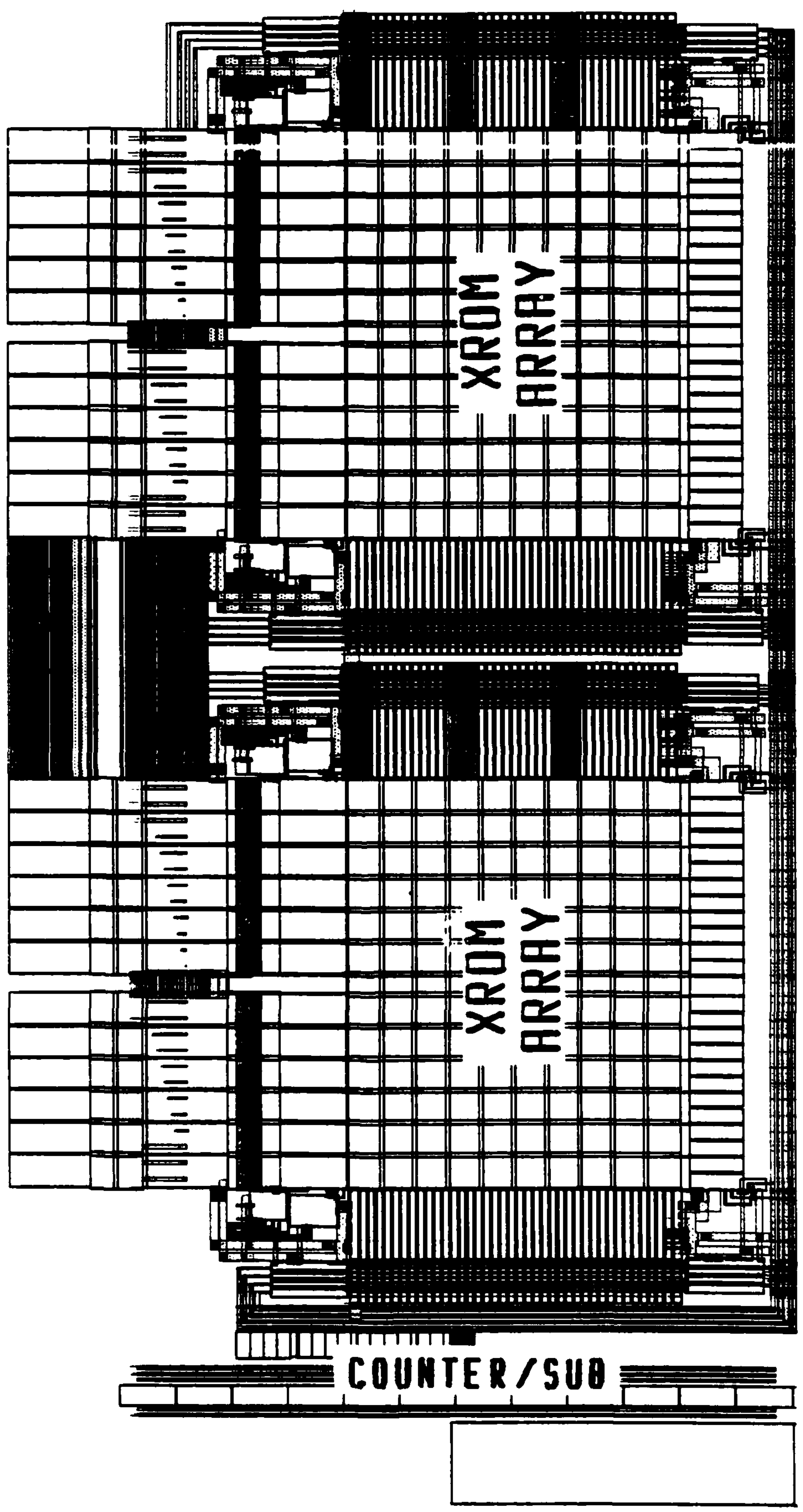


Figure III-15 XROM Address Generator

proper MSFF bank (input or output). The 48 bits comprise 4 sequential data addresses that are multiplexed out to the 12-bit input and output data address buses from the input and output MSFF banks respectively.

The counter/subtractor circuit alternates the addresses applied to the XROM to retrieve input addresses and output addresses in an alternating pattern. The number of delays in the WFTA16 pipeline dictates that the output address sequence be an exact duplicate of the input address sequence except delayed by 59 addresses. Since four addresses are retrieved per XROM access and output addresses are delayed by two, the subtractor subtracts a constant of 14 from the input address counter and 2 MSFF delays are added to obtain the correct output data addresses. Thus, when the counter's value and subtractor's output are alternated, the XROM output alternates between input and output data addresses. The set of outputs are routed and multiplexed to the corresponding I/O address bus to obtain both address sequences. On start up of the WFTA, false data may be written out to bogus address locations until the pipeline is filled. However, this data will eventually be overwritten.

The speed of the XROM is improved through the use of precharging and output sense amplifiers. When Precharge rises, new addresses flow into the PLA decoder, 4 to 1 multiplexer, and address columns. The information flows as far as the wordlines from the PLA decoder, the one channel of four is selected, and the address columns are driven. Concurrently, all bitlines and sense outputs are precharged high. During Precharge', the appropriate bitlines and word sign bitlines are pulled low through the array devices. The outputs are latched near the end of Precharge'. This scheme results in simple and fast address generation from the XROM.

Another technique employed to increase the XROM's speed of operation is to break the array vertically into smaller blocks. This technique decreases the length, resistance, and capacitive loading of the wordlines. It does, however, sacrifice some area since the PLA decoder circuitry that drives the wordlines must be duplicated.

The XROM Address Generator shown in Figure III-15 and described briefly above has a number of circuit modules that deserve further explanation. These descriptions of specific cell operation will be delayed and presented with the schematics and cell descriptions given in the "XROM cells" section.

Testability Considerations. The discussion and techniques presented in the Control Sequencer Testability Considerations section apply here as well. All three techniques used for the Control Sequencer can also be used on the XROM to increase the testability of the circuit. Probe points were placed as often as possible to achieve greater observability of the XROM circuit (although placing probe points within the dense XROM array was not practical). The XROM output MSFFs and counter/subtractor MSFFs were serially chained together so the proper addresses could be controlled or observed from test pins. Test vectors can be loaded into the counter/subtractor MSFFs, and the XROM output can be read from the output MSFFs. Finally, the XROM clocks were not connected to the same pins as the arithmetic circuitry. All of these techniques (presented in the Control Sequencer Section) were designed into the XROM circuitry.

Another technique that would be useful during WFTA chip testing is to control the input and output data addresses externally. If the XROM circuit did not function properly during a WFTA chip test, the arithmetic and Control Sequencer sections could still be successfully tested by externally applying the correct address sequence to the input and output data RAMS.

Although all the above techniques are useful, the key to success is insuring that an operational XROM test chip is produced before fabricating the WFTA chips.

XROM Address Generator Cells. The cell descriptions for the XROM Address Generator circuit are given in the paragraphs below. The cell presentation will be given in the same order as the signal information flows through the circuit. The placement of the cell in the overall XROM circuit can be seen by referring to Figure III-15.

Load State PLA. The Load State PLA is used to initialize the XROM circuit for a particular DFT size as determined by the two-bit DFT size variable externally input to the WFTA chip. The Load State PLA circuit is depicted in Figure III-16.

The two-bit DFT variable externally loaded in the WFTA16 chip is stored in the two-bit state register. Table III-1 shows each of the four 2-bit DFT variables and its corresponding DFT size for the 15, 16, and 17 point WFTA chips.

DFT VARIABLE	WFTA PROCESSOR SIZE		
	15	16	17
0 0	4080	4080	4080
0 1	15	16	17
1 0	255	272	255
1 1	240	240	272

Table III-1 DFT Sizes for WFTA Processors.

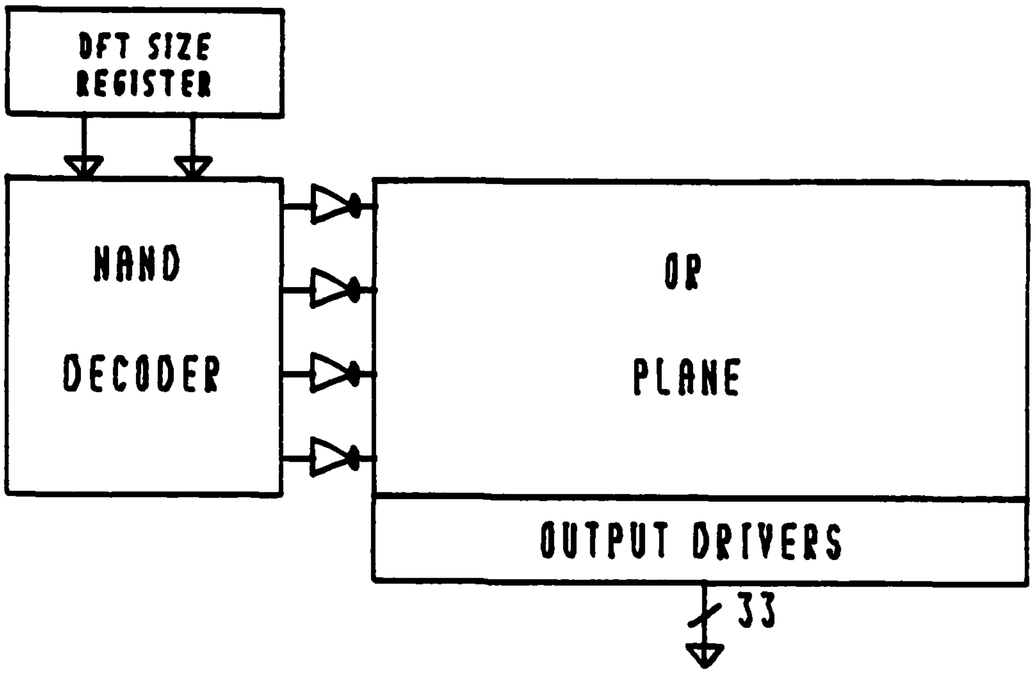


Figure III-16 Load State PLA

The outputs of the state register are input to the AND-plane of the Load State PLA. For each DFT size, a corresponding start counter value, last input XROM address value, and last output XROM address value are output from the PLA in a 33-bit wide word. The PLA outputs remain stable throughout the calculation of the DFT. The start counter value is loaded into the XROM addressing counter in order to start the address output sequence at the correct location in the XROM for that DFT size. The last address values are used to generate the DONE-IN and DONE signals that are used for external interfacing and stopping of the WFTA processor.

The PLA is made up of the same cells used in the Control Sequencer PLA.

Counter/Subtractor. The Counter/Subtractor cell performs two main functions. It generates the XROM address and it outputs the DONE-IN and DONE signals. This Counter/Subtractor circuit is depicted in Figure III-17.

The counter is loaded from the start counter value output of the Load State PLA. The counter output is latched into an output buffer to allow the counter to operate while the last counter outputs are still stable. The counter output is applied to the XROM for four clock cycles when the IN/OUT signal is high. During the next four clock cycles (when IN/OUT is low), the subtractor circuit's output is applied to the XROM. This alternating scheme is continued to output both input and output data addresses from the XROM.

After each of the eight clock cycles needed to address the XROM with both input and output addresses, the counter's new incremented value is latched into the output buffer for the next set of accesses. The control signals (Counter Clock, LOAD, INC, and IN/OUT) are generated by the Control Sequencer.

While the circuit is addressing the XROM, two comparator circuits are operating to flag the last input access and the last output access. The DONE-IN flag is set when the last input data value has been read and the input memory

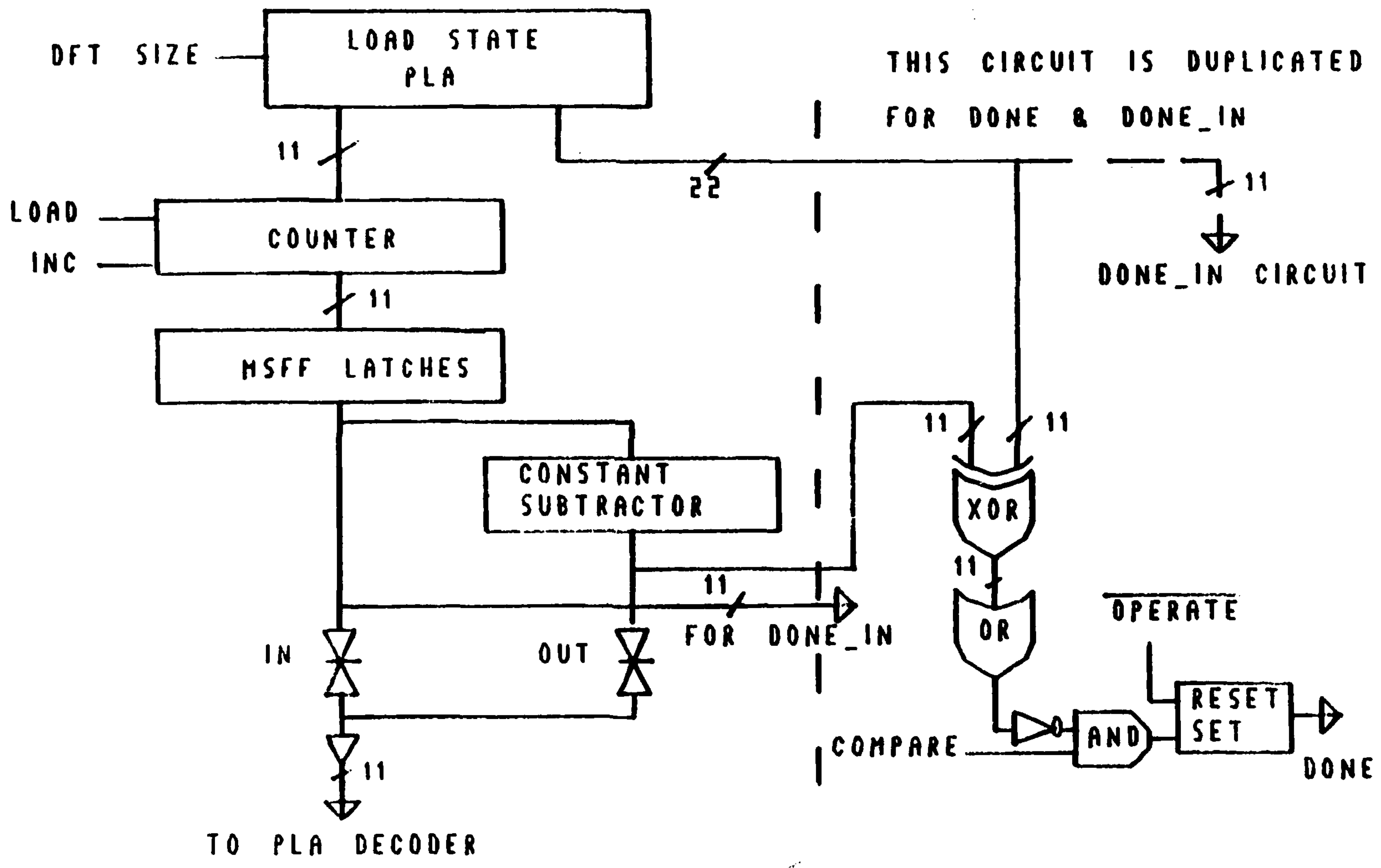


Figure III-17 Counter/Subtractor Circuit

is free to be released by the Interface chip. The DONE flag is set when the last output input data value has been written to the output memory. This flag stops the WFTA operation by inhibiting the Write Strobe signal immediately after writing the last data word to the output memory, and by stopping the bit in the Control Sequencer ring counter from cycling back to the first MSFF in the chain. The DONE signal also informs the Interface Chip that the assigned DFT has been completed.

The counter and subtractor cells must be capable of operating at speeds 8 times and 4 times slower than the system clock respectively.

A loadable counter capable of operating at these speeds was designed and implemented. The counter bit-cell and LSB cell are shown in Figure III-18 and Figure III-19 respectively. The counter is a loadable, asynchronous, binary counter that receives LOAD and INC control signals as input. When LOAD is high for two clock cycles the counter is loaded with the value applied to the load lines. The normal operation of the counter is controlled by the INC control signal. An examination of the counter cells will show that the worst-case "roll-over" can occur in one clock cycle provided that the clock rate is slow enough to allow the ripple carry to travel through 11 transmission gates with a driver between each.

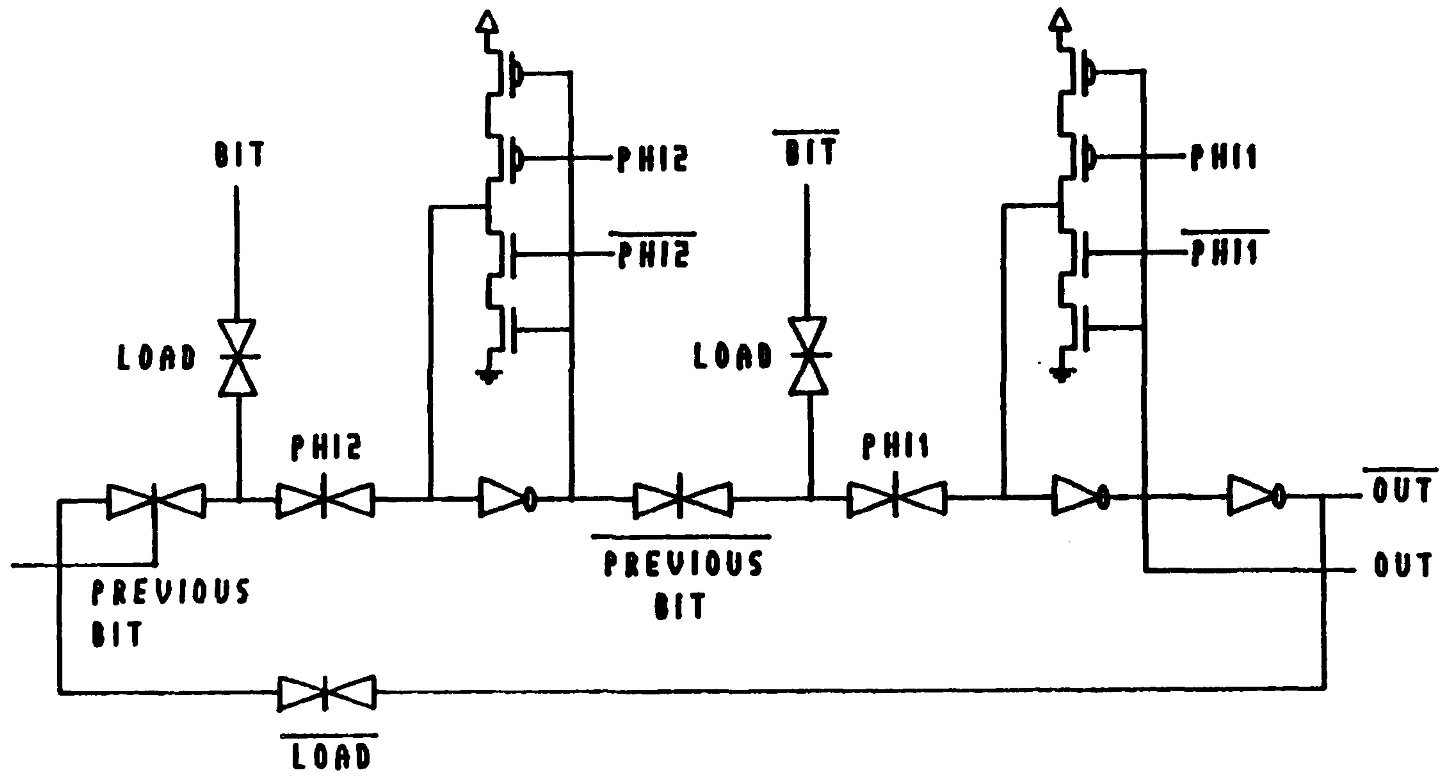


Figure III-1B Counter Bit-Cell

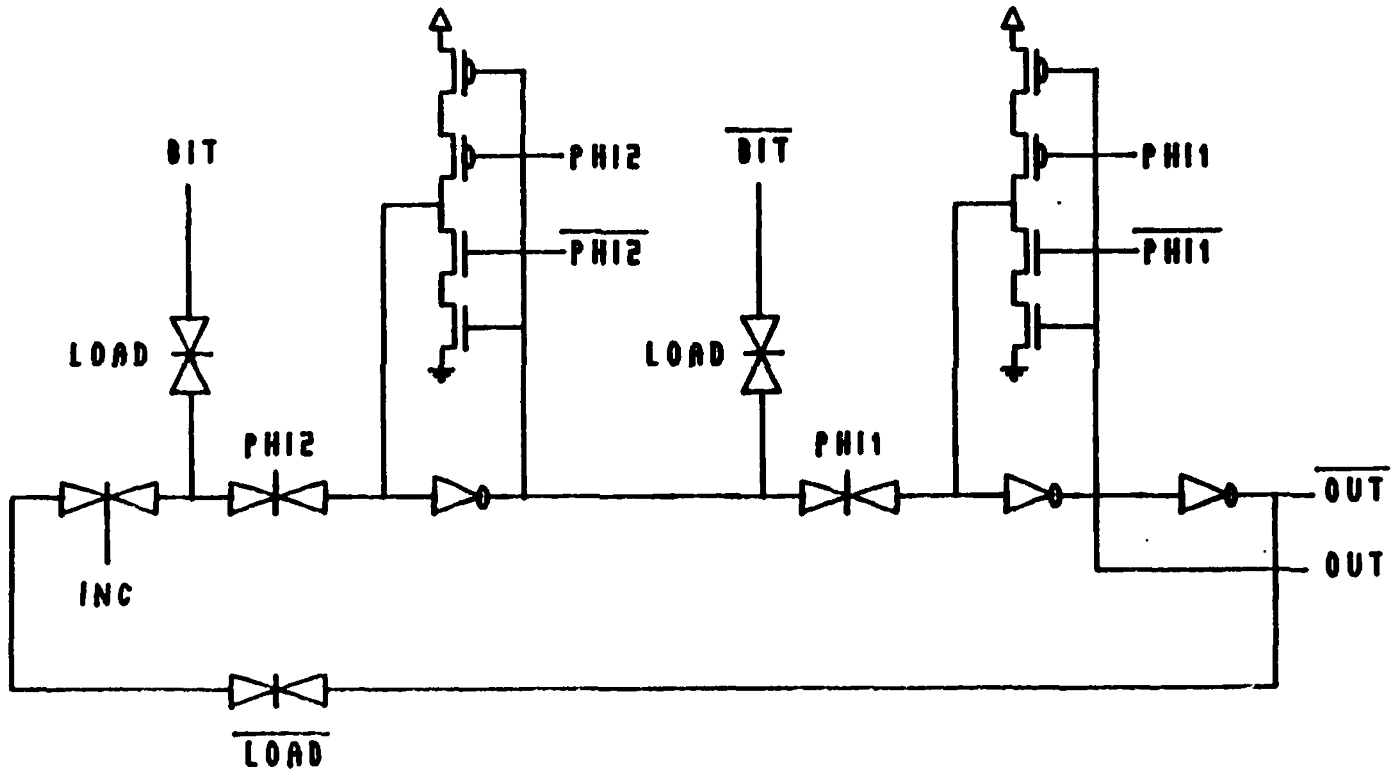


Figure III-19 Counter LSB-Cell

The subtractor cells were designed to operate at the speeds required by capitalizing on the fact that the subtractor will always subtract a hard-wired constant. Since the constant is designed into the circuit, each normal full-subtractor bit becomes a type of half-subtractor.

The Boolean equations for a full-subtractor's borrow and difference are:

$$B = x'y + x'z + yz$$

$$D = x'y'z + x'yz' + xy'z' + xyz.$$

If x is the bit input, y is the fixed constant, and z is the borrow from the previous bit, the equations reduce to

$$B = x'z$$

$$D = x'z + xz' = x + z$$

for a fixed-0 subtract, and

$$B = x' + z$$

$$D = x'z' + xz = x \cdot z$$

for a fixed-1 subtract.

These equations can be simply implemented as shown in Figure III-20 for both fixed-0 and fixed-1 subtractor cells.

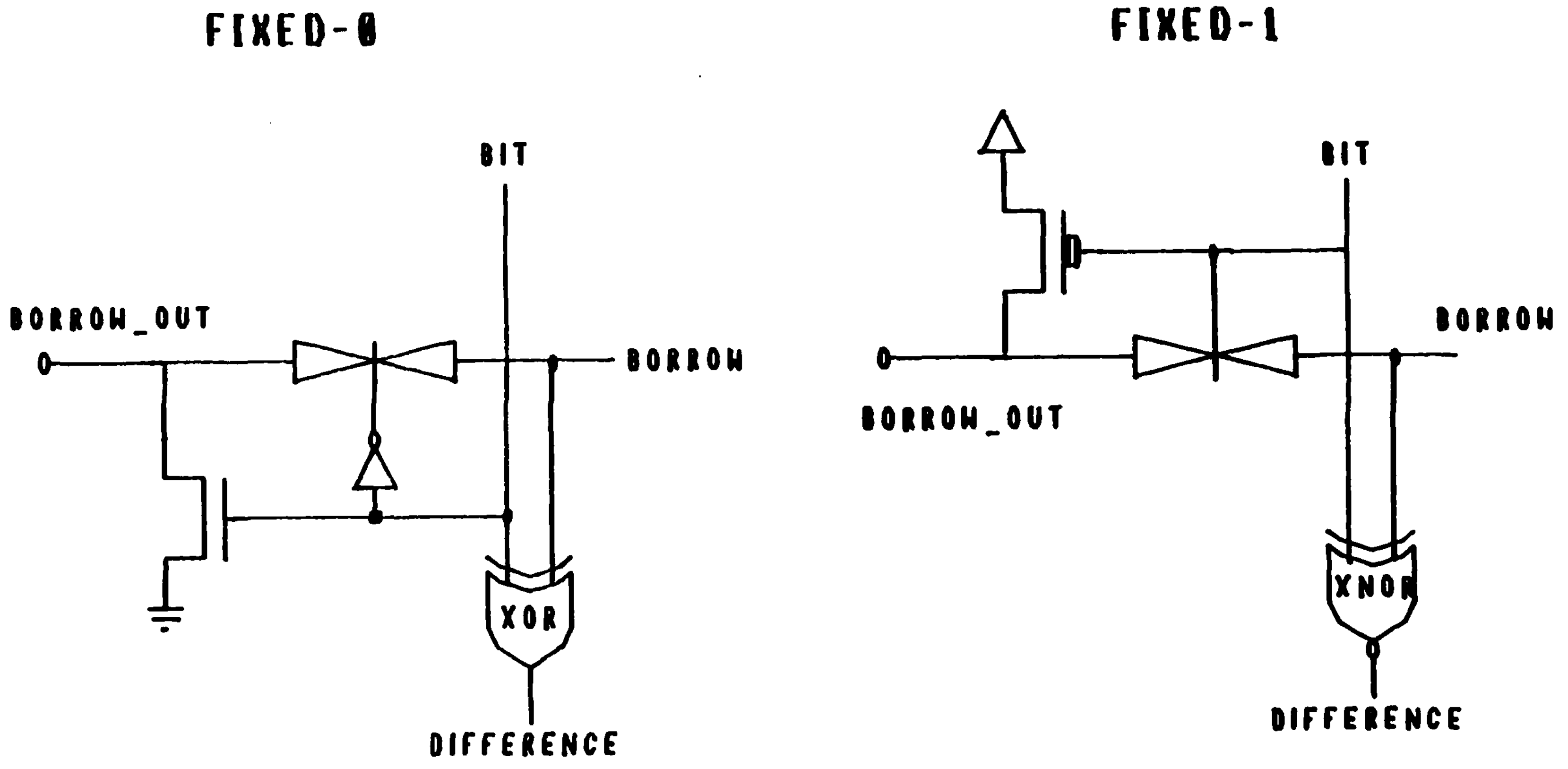


Figure III-20 Subtractor Cells

The borrow and difference outputs are obtained with only one transmission-gate delay. The borrow signal that is propagated down through the subtractor is buffered by a double inverter every four bit cells to improve the circuit's speed since the delay is proportional to the square of the number of series devices (Mead and Conway, 1980:22,23).

The last address comparator circuits are implemented as shown in Figure III-17. Each bit of the counter or subtractor output is XORed with the last address value from the Load State PLA. These 11 XOR outputs are ORed together and this signal is input to a SRFF after being ANDed with a COMPARE control signal. Thus, the DONE or DONE-IN SRFF will be set only if the last address was stable on the subtractor or counter outputs respectively. The DONE-IN and DONE flags are delayed so as to not halt processing before the last address (input or output) is placed on the data address bus for two clock cycles.

XROM PLA Decoder. The XROM PLA decodes the 8-bit address input and raises the wordline that corresponds to that address. The XROM wordline decoder is a NAND decoder that provides a regular and compact structure. This structure can be easily placed along the side of the XROM array and allows the decoder personalizations to be placed in any order desired. The former characteristic saves chip area and the latter allows the easy implementation of ROW swapping discussed in Chapter IV.

The use of a "snaked" gate on the series NAND devices provides for large device width with a small diffusion area. The vertical pitch constraint of the decoder cells is very small to match the 12 lambda height of the XROM cell, and the series devices must fit within this height. By having two decoder cells share contacts to the vertical input and input' lines, and personalizing the decoder cells in polysilicon, additional space is saved for greater active area widths. A double NAND decoder cell's CIF-plot is shown in Figure III-21.

Wordline Driver/Pullup. Normally, the decoder's output to the wordline is pulled high when it is not the selected wordline. The decoder's output is input to the XROM's wordline through three staged-up inverters. The three inverter's invert the decoder's output signal and provide for fast switching of the XROM's wordline. The static p-device pullup and the three wordline driving inverters are layed out in the same 12 lambda vertical pitch per wordline as the decoder was. The CIF-plot of the pullup and first of the three inverters is shown in Figure III-22.

The schematic for a wordline slice of the XROM NAND PLA decoder and the wordline driver is shown in Figure III-23.

XROM Array. The XROM array (briefly introduced in Chapter II) is the means by which all of the proper

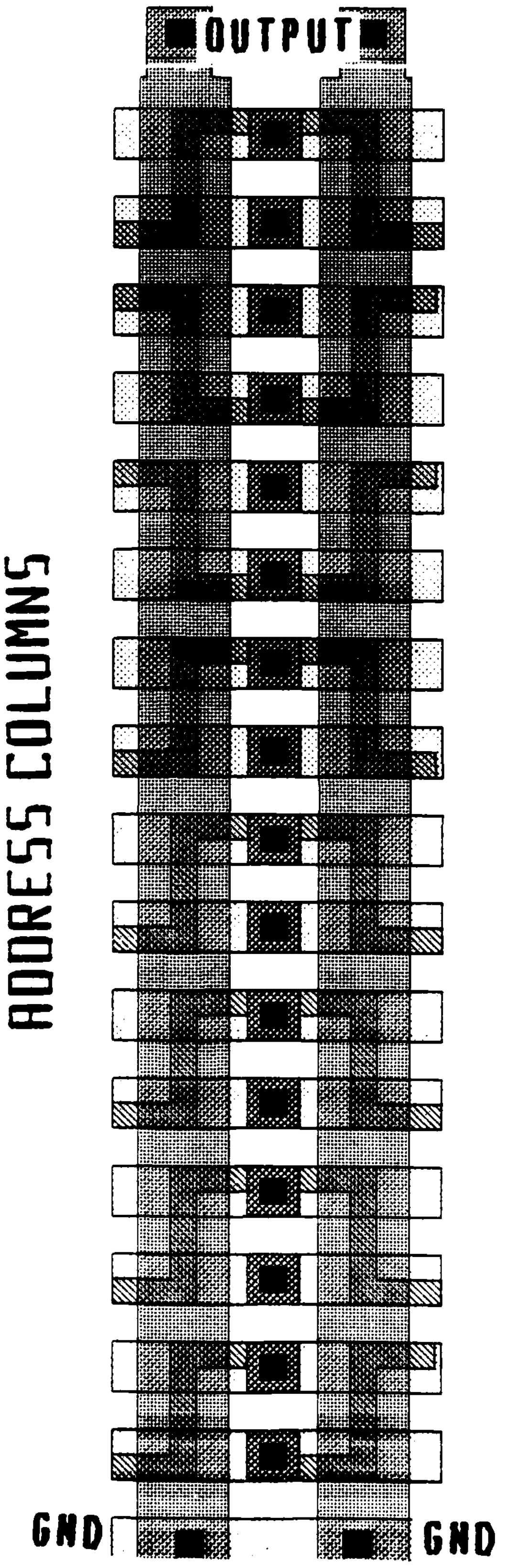


Figure III-21 Double NAND Decoder

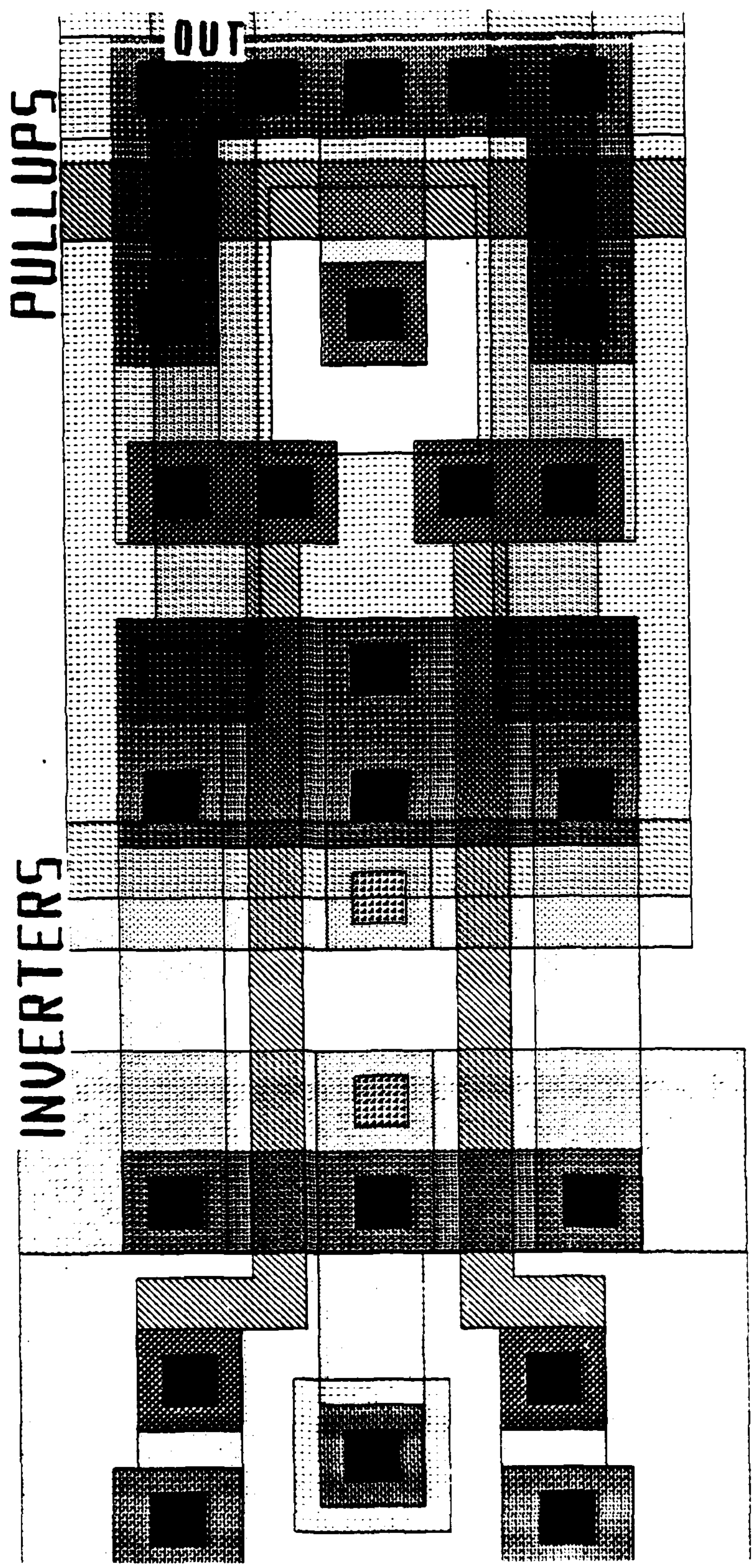


Figure III-22 Pullup/Driver Circuit

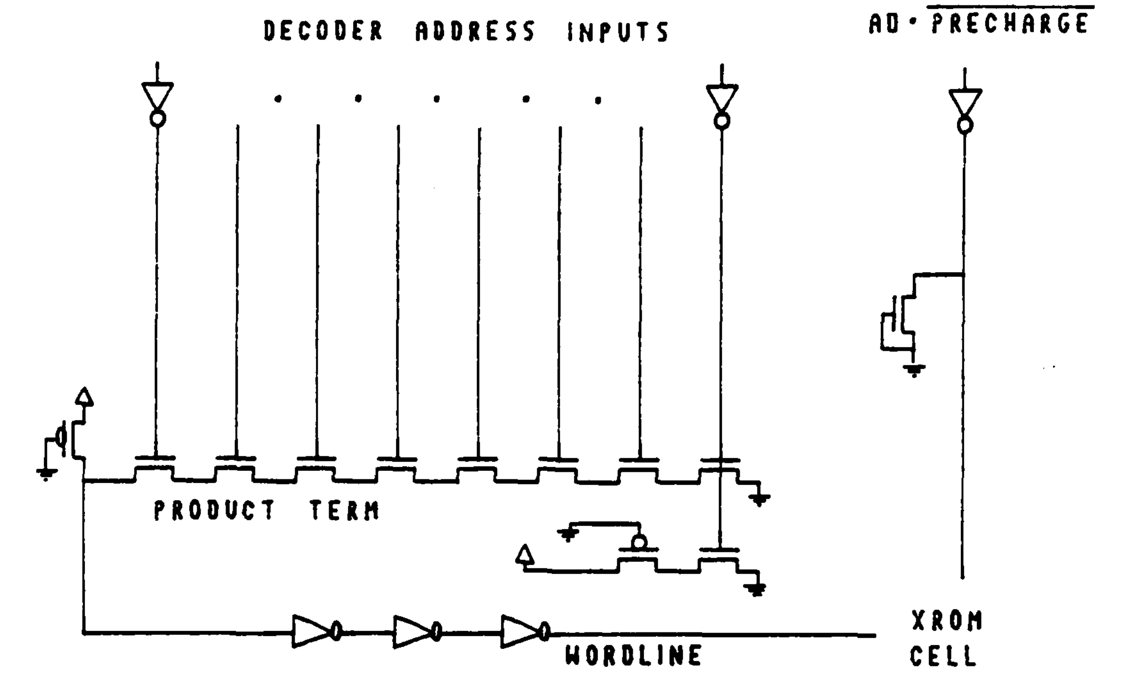


Figure III-23 XROM Decoder Schematic

data address sequences are stored in a small area. The XROM cell is the key element of the XROM array, and must be understood to appreciate the array's operation.

The XROM cell receives its name from the pattern of devices which resemble the letter X as can be seen from Figure III-24. The column address lines (labeled PRECH+A0 and PRECH+A0' in Figure III-24) are driven high and low when Precharge is low. At that point, the wordline of the XROM that is driven high has already turned on any n-devices whose gates are on that wordline. If the device's source is connected to a column address line that is selected and pulled low the bitline will be pulled low. If the device's source is

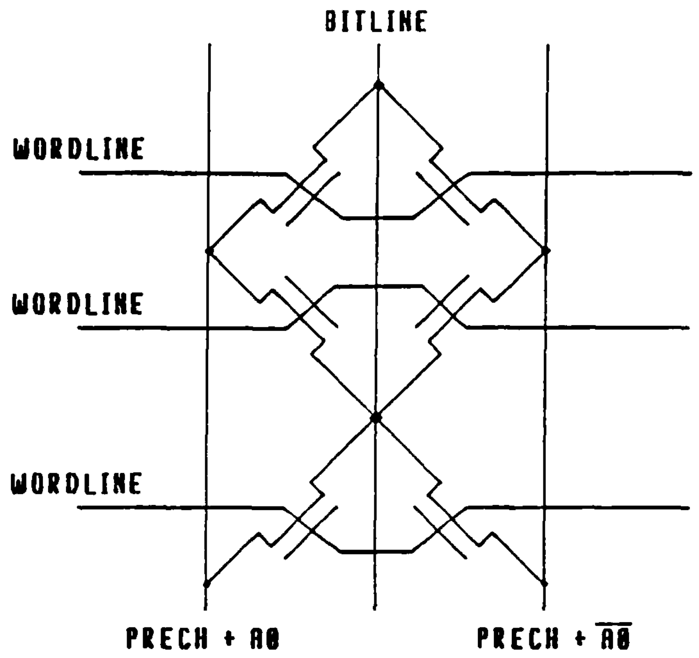
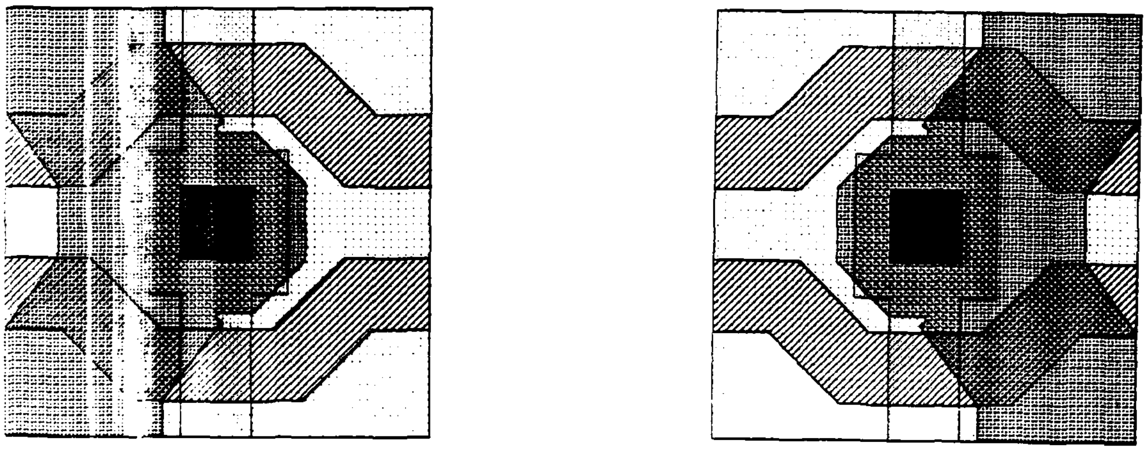


Figure III-24 XROM Cell Schematic

connected to a column address line that is not selected (and is high) or if no devices are connected to the bitline (because of the programming), the bitline is not pulled low. If a device is connected to both address lines, the bit line will still be pulled low since the n-device will pass a low voltage more strongly than a high voltage. Additionally, the bitline voltage only needs to drop below 3.3 volts (approximately) for the sense amplifier to "sense" a zero.

The active area personalizations on each wordline determine the data word output on the bitlines. Each XROM cell stores four bits, two (one for A0, one for A0') for each wordline that runs through it. A zero is represented by no active area personalization and a one is achieved by placing active area to create an n-device. Figure III-25 show two of the 16 possible cell configurations.



a.) A0 bits b.) A0' bits

Figure III-25 Two possible XROM cells.

Figure III-25-a shows a "one" personalization for both A0 bits in the cell and Figure III-25-b for both A0' bits. The XROM array shown in Figure III-26 is made up of cells that have four one bits or four zero bits. An improved cell is described for the four zero bit cell in Chapter IV.

The WFTA16 XROM array outputs a word that is 192 data bits and 4 word sign bits wide. The word sign bits are used to invert portions of the data words output (see Chapter IV). The 192 data bits are multiplexed to 48 bits by the 4 to 1 multiplexer contained in the sense amplifier/multiplexer cell. Eight of the XROM addresses are input to the PLA wordline decoder and one is input to the address column lines that run vertically between bitlines in the XROM array.

The XROM array is built from XROM cells properly placed on the bit and column address lines. The 8-bit address applied to the PLA wordline decoder determines which wordline is driven high and able to turn on its n-devices. The address bit input into the address column lines determines which side (left or right) of the bitlines' personalization will be output on the bitlines. The 2-bit address input to the 4 to 1 multiplexer then selects the one bitline of four in a "column-byte" to be output through the sense amplifier. This process reduces the 192-bit data word down to the desired 48-bit data word. The four word sign bitlines placed in the middle of the arrays are always selected and are used in the sense amplifier cells. A section of the XROM array is shown in Figure III-26.

Multiplexer/Sense Amplifier. The Multiplexer/ Sense Amplifier cell selects one of four XROM bitline outputs, and outputs that bitlines' data or inverse

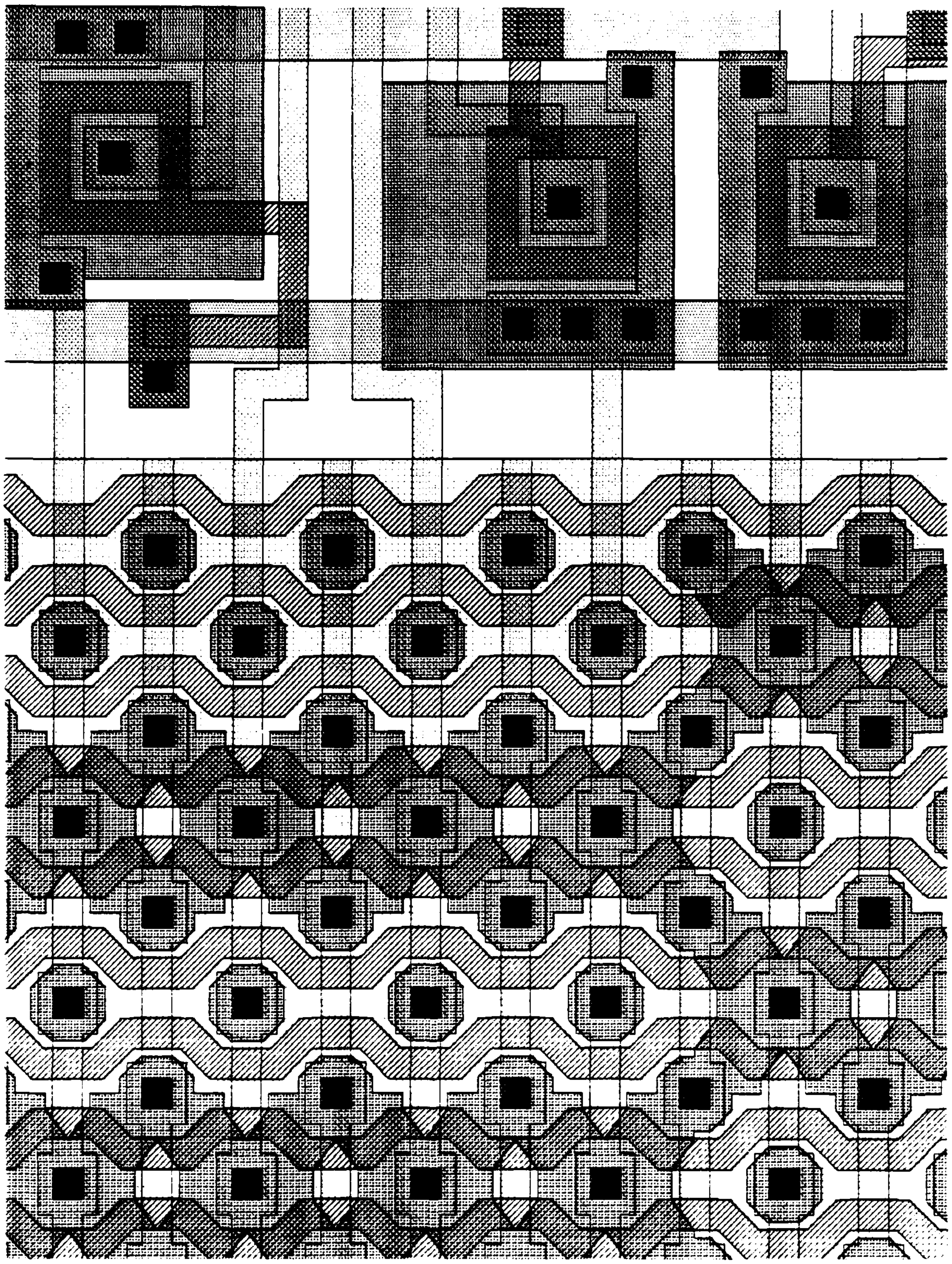


Figure III-26 XROM Array Section

depending on the column sign bit and word sign bit. The sense amplifier speeds the operation of the XROM circuit. Figure III-27 depicts the schematic of a non-inverting multiplexer/sense amplifier circuit. The inverting circuit (used when the entire column's data is to be inverted, see Chapter IV) is the same except the word and word' inputs to the transmission gates are reversed.

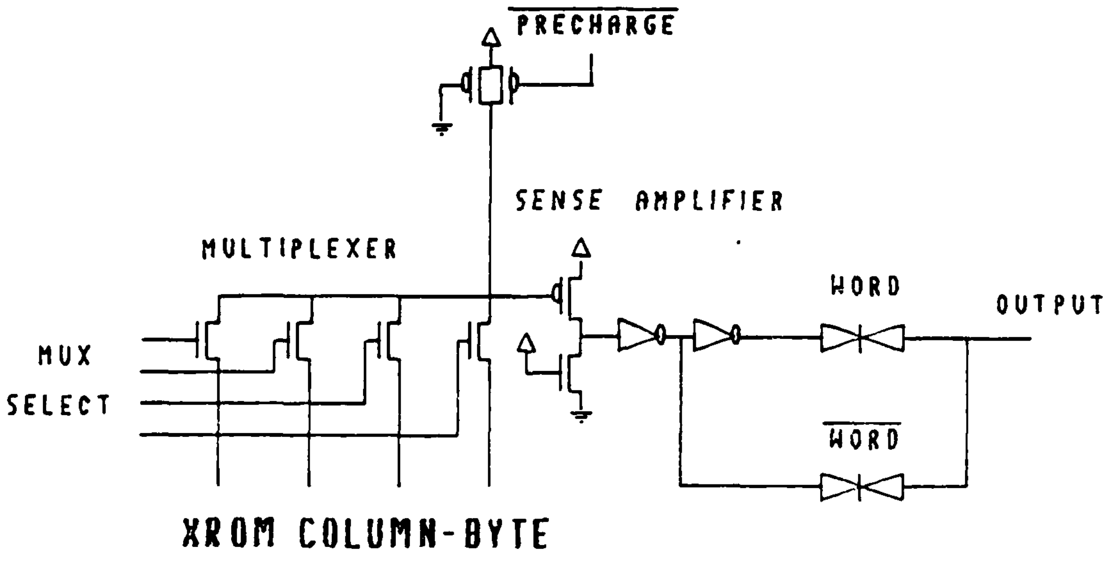


Figure III-27 Non-inverting Multiplexer/Sense Amplifier Schematic

The multiplexer portion of the circuit is implemented with the four n-devices that select the one of four bitlines. The n-device whose gate is driven high by the 2 to 4 decoder allows its bitline information to pass into the sense amplifier portion of the circuit.

The sense amplifier operates in the following manner. The bitline from the XROM array will be approximately 3.3 volts when not pulled low because the bitlines are precharged high through the multiplexer's n-device. At this voltage the sense line will not drop below the 5-volt precharged level since the n-device of the multiplexer is between the bitline and sense line. As the high capacitance bitline is slowly pulled low by a n-device of the array, the n-device of the multiplexer will be able to rapidly discharge the small capacitance of the sense line. With the sense inverter designed to switch at approximately 4 volts, the bitline's state is quickly "sensed". Figure III-28 shows the voltage traces of the bitline and sense line during a read of a one data bit.

One of the keys to the sense amplifier operation is keeping the capacitance on the sense line small while still having large current drive through each of the four multiplexer devices. The solution to this problem was to use a donut device for each of the multiplexer devices. The donut devices provided large current drive with minimal drain and sidewall capacitances on the sense line. The

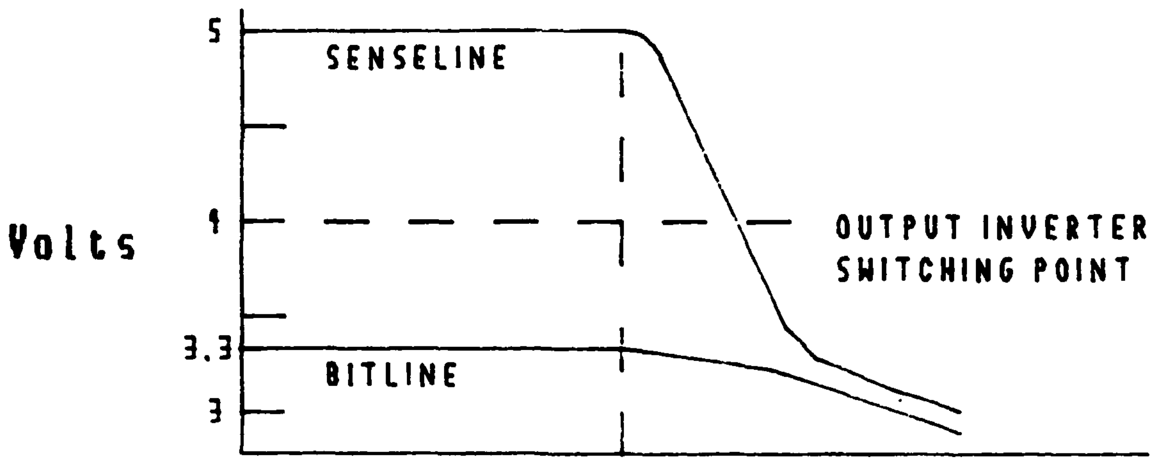


Figure III-28 Sense Amplifier Operation.

donut devices can be seen in Figure III-26.

Word Sign Bit Sense Amplifier/Driver. The word sign bit sense amplifier/driver cell operates in a manner very similar to the multiplexer/sense amplifier cell described above. The differences are that no multiplexer is needed, the output is never inverted, and greater output drive is needed. The circuit's schematic is shown in Figure III-29.

The cell handles two word sign bitlines by duplicating the circuit shown in Figure III-29. The cell and word sign bits are placed in the middle of each XROM array half and control the word and word' inputs to the column sense amplifiers. A word sign bit output controls a 12 column-byte group, and its output must be used to determine which state (x or x') of the sense amplifier signal is to be output for that group. The word sign bit sense amplifier

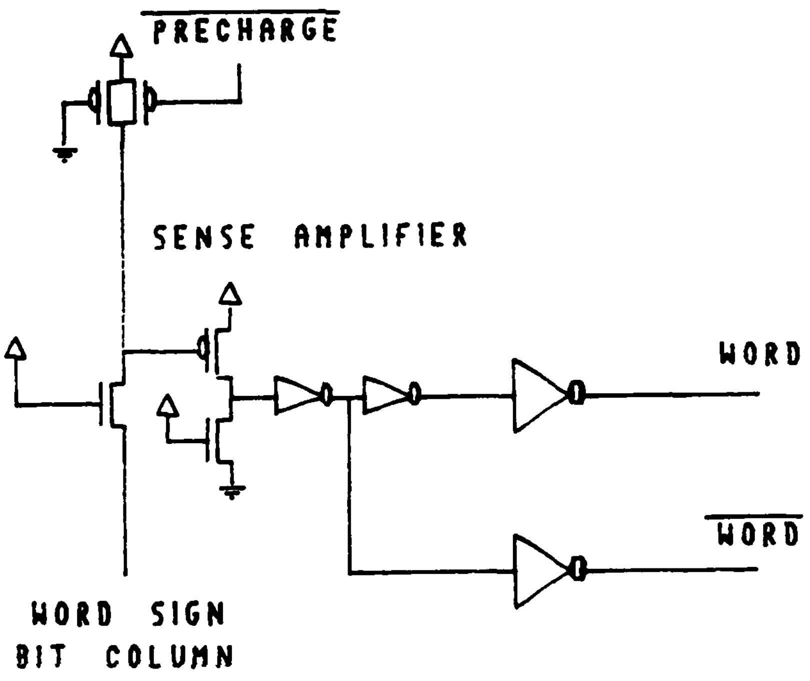


Figure III-29 Word Sign Bit Sense Amp.

circuit can operate faster than the column sense amplifiers which allows the word and word' signal to arrive in time. The improved speed is a result of the reduced capacitance on the sense line.

<u>MSFF Banks/Final Multiplexer</u>. At this point the data addresses output from the XROM need only be routed to the proper MSFF bank (input or output) and multiplexed out

to the input or output data address bus. Both the input and output bank of MSFFs are connected to the XROM outputs. The XROM output signal is passed through the input MSFF bank in first metal in order to reach the output MSFF bank placed as shown in Figure III-15.

The input and output MSFFs have transmission gates on the inputs. The XROM outputs are shifted into the proper MSFF bank by the SHIFT_UP-IN and SHIFT_UP-OUT control signals generated by the Control Sequencer.

Once the 48 bits retrieved from the XROM are stored in the proper MSFF bank, a multiplexer for each bank outputs the four 12-bit addresses to the appropriate data address bus every two clock cycles. The multiplexer circuit is simply a transmission gate controlled by a 4-bit ring counter on each MSFF output path to the address bus.

<u>SPICE Simulations</u>. The worst-case path of the XROM address generation circuitry was simulated using the SPICE program. The schematic of the worst-case path through the XROM is shown in Figure III-30.

This path can be broken up into two sections for the sake of presentation. The first section consists of the addresses being input to the PLA decoder and the wordlines changing state. The second section uses the wordline signal as the input to the XROM devices which changes the bitline and sense amplifier output. The time to traverse each path separately can be summed to determine the total XROM speed.

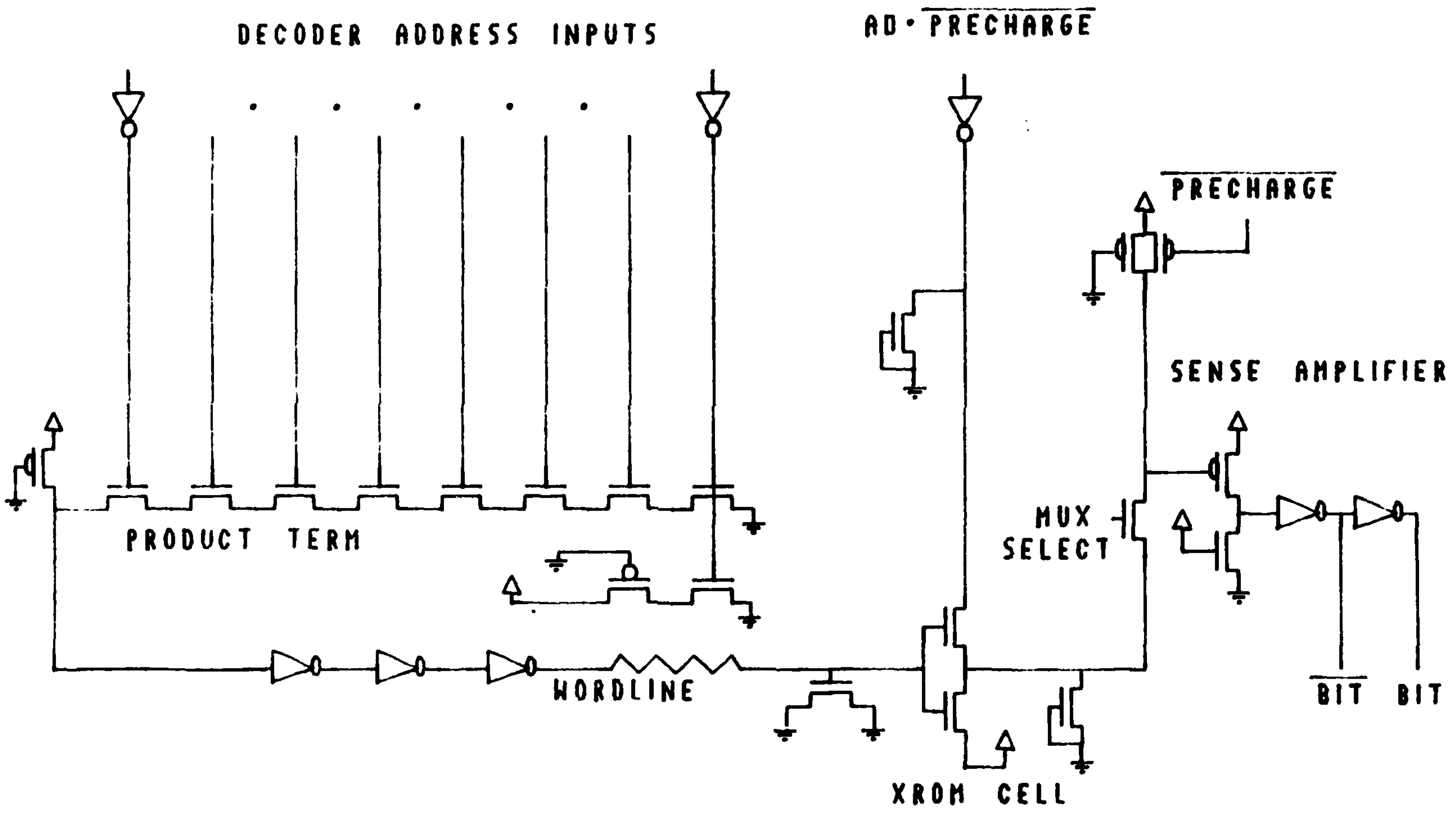
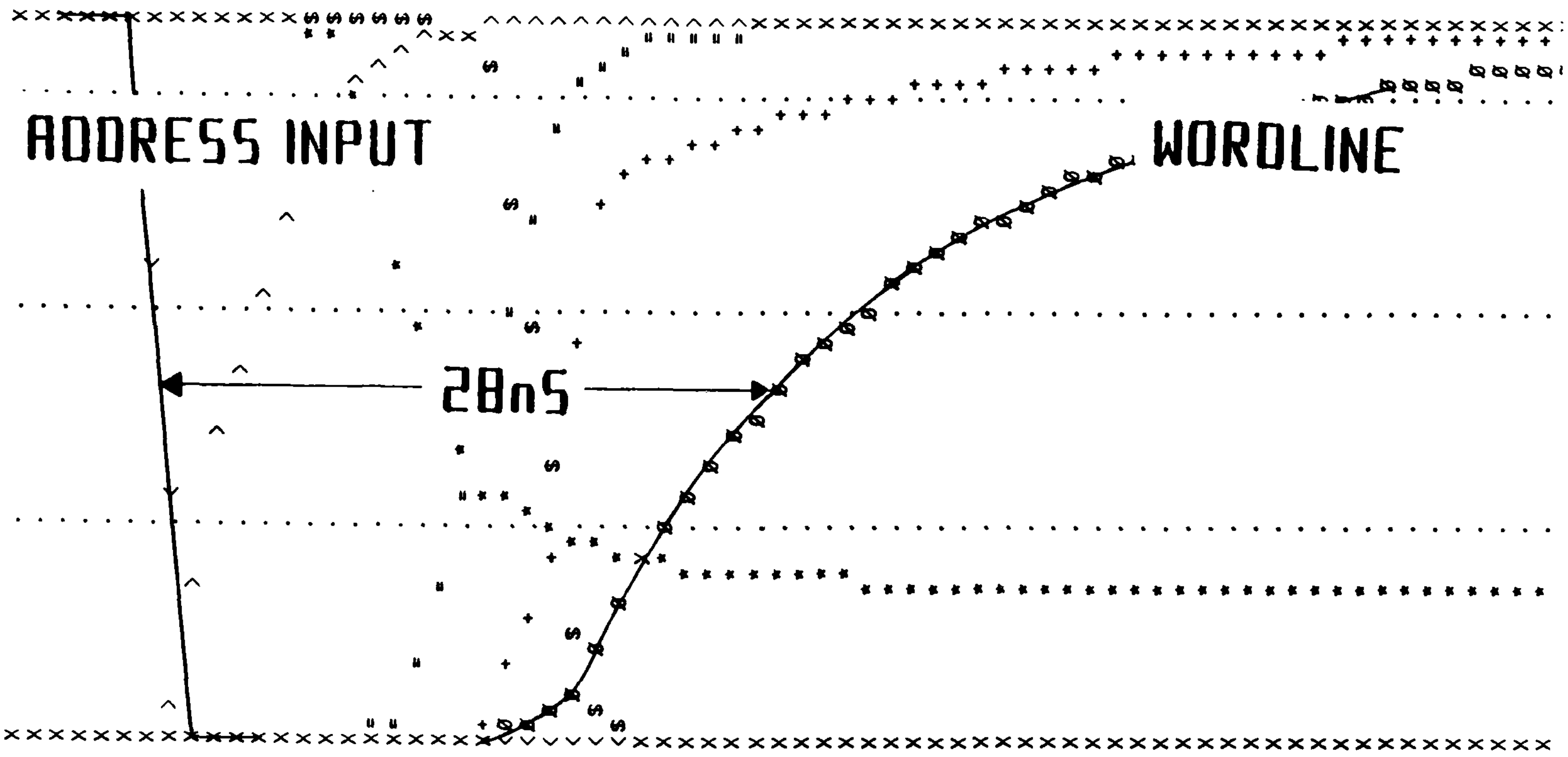

DECODER ADDRESS INPUTS
AD · $\overline{\text{PRECHARGE}}$
$\overline{\text{PRECHARGE}}$
SENSE AMPLIFIER
PRODUCT TERM
MUX SELECT
WORDLINE
$\overline{\text{BIT}}$ BIT
XROM CELL

Figure III-30 XROM Worst-Case Path

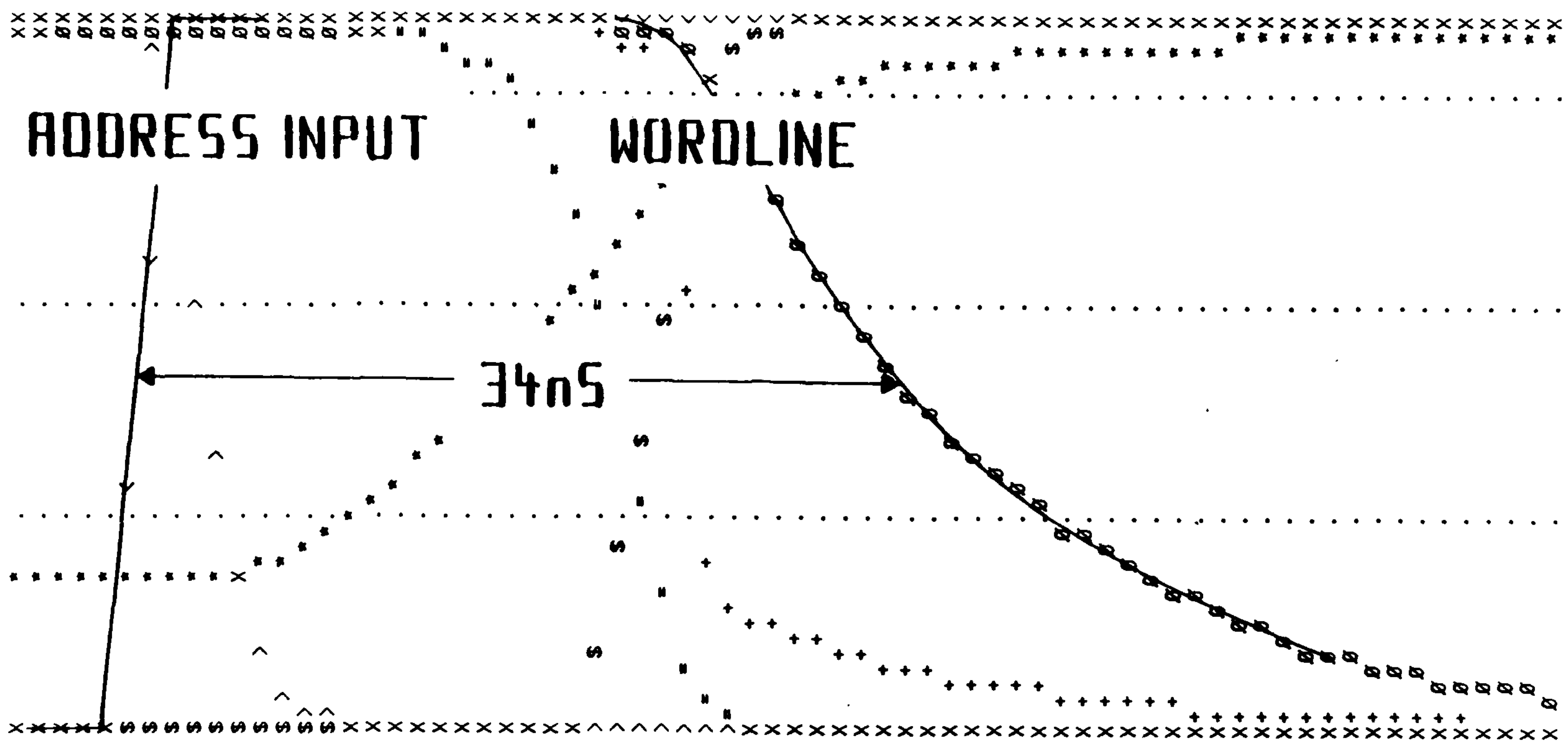
Figures III-31 and III-32 show the timing for a wordline being selected and unselected respectively. The worst-case time for the wordline transition is less than 35 nS after the address is applied to the decoder. Figures III-33 and III-34 show the timing for an output going high and returning low respectively. The Precharge cycle returns the output low while the output must go high during the Precharge' cycle. As the SPICE simulations show, both of the output transitions can occur within 25 nS of the wordline changing state.

Thus a data word can be retrieved from the XROM in 60 nS (35 nS Precharge time) for a 3 micron process and a SPICE model as given in Appendix E. This means that four data address words can be output at a rate of over 65 MHz per word. Since input and output data addresses must be output to the bus simultaneously, the XROM must perform two accesses. This effectively cuts the rate in half to just over 30 MHz per data word pair. Since a new input and output data address pair is needed only every other clock cycle, the system clock can run at a rate of over 65 MHz. This figure is compatible with the 55 + MHz operating speed of the Control Sequencer. In fact, since the Precharge signal is generated by the Control Sequencer, it can be produced from a quarter frequency clock (as long as the clock speed is less than 60 MHz for the 3 micron process). Additionally, a speed increase is expected for a 1.2 micron process implementation of the XROM circuit as was expected for the 1.2 micron process implementation of the Control Sequencer.



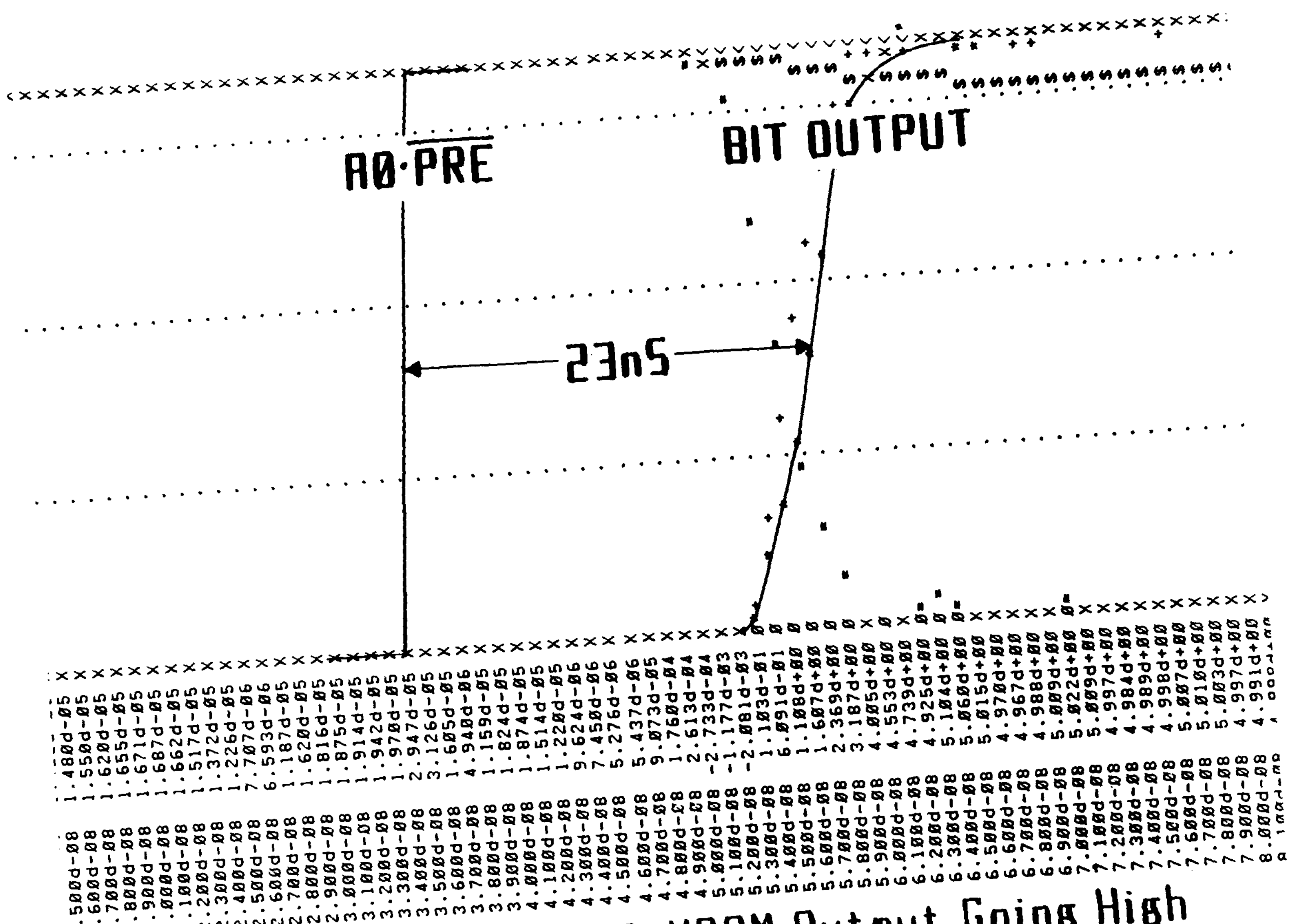
```
Ø.        d+ØØ   5.ØØØd+ØØ
1.ØØØd-Ø9   5.ØØØd+ØØ
2.ØØØd-Ø9   5.ØØØd+ØØ
3.ØØØd-Ø9   5.ØØØd+ØØ
4.ØØØd-Ø9   5.ØØØd+ØØ
5.ØØØd-Ø9   5.ØØØd+ØØ
6.ØØØd-Ø9   5.ØØØd+ØØ
7.ØØØd-Ø9   5.ØØØd+ØØ
8.ØØØd-Ø9   5.ØØØd+ØØ
9.ØØØd-Ø9   5.ØØØd+ØØ
1.ØØØd-Ø8   5.ØØØd+ØØ
1.1ØØd-Ø8   5.ØØØd+ØØ
1.2ØØd-Ø8   5.ØØØd+ØØ
1.3ØØd-Ø8   4.981d+ØØ
1.4ØØd-Ø8   4.921d+ØØ
1.5ØØd-Ø8   4.471d+ØØ
1.6ØØd-Ø8   3.9Ø4d+ØØ
1.7ØØd-Ø8   3.337d+ØØ
1.8ØØd-Ø8   2.873d+ØØ
1.9ØØd-Ø8   2.464d+ØØ
2.ØØØd-Ø8   2.Ø54d+ØØ
2.1ØØd-Ø8   1.733d+ØØ
2.2ØØd-Ø8   1.669d+ØØ
2.3ØØd-Ø8   1.6Ø4d+ØØ
2.4ØØd-Ø8   1.54Ød+ØØ
2.5ØØd-Ø8   1.446d+ØØ
2.6ØØd-Ø8   1.387d+ØØ
2.7ØØd-Ø8   1.338d+ØØ
2.8ØØd-Ø8   1.296d+ØØ
2.9ØØd-Ø8   1.261d+ØØ
3.ØØØd-Ø8   1.232d+ØØ
3.1ØØd-Ø8   1.2Ø9d+ØØ
3.2ØØd-Ø8   1.19Ød+ØØ
3.3ØØd-Ø8   1.171d+ØØ
3.4ØØd-Ø8   1.154d+ØØ
3.5ØØd-Ø8   1.143d+ØØ
3.6ØØd-Ø8   1.133d+ØØ
3.7ØØd-Ø8   1.125d+ØØ
3.8ØØd-Ø8   1.119d+ØØ
3.9ØØd-Ø8   1.113d+ØØ
4.ØØØd-Ø8   1.1Ø9d+ØØ
4.1ØØd-Ø8   1.1Ø5d+ØØ
4.2ØØd-Ø8   1.1Ø2d+ØØ
4.3ØØd-Ø8   1.Ø99d+ØØ
4.4ØØd-Ø8   1.Ø97d+ØØ
4.5ØØd-Ø8   1.Ø95d+ØØ
4.6ØØd-Ø8   1.Ø94d+ØØ
4.7ØØd-Ø8   1.Ø93d+ØØ
4.8ØØd-Ø8   1.Ø91d+ØØ
4.9ØØd-Ø8   1.Ø9Ød+ØØ
5.ØØØd-Ø8   1.Ø9Ød+ØØ
5.1ØØd-Ø8   1.Ø89d+ØØ
5.2ØØd-Ø8   1.Ø89d+ØØ
5.3ØØd-Ø8   1.Ø88d+ØØ
5.4ØØd-Ø8   1.Ø88d+ØØ
5.5ØØd-Ø8   1.Ø88d+ØØ
5.6ØØd-Ø8   1.Ø87d+ØØ
5.7ØØd-Ø8   1.Ø87d+ØØ
5.8ØØd-Ø8   1.Ø87d+ØØ
5.9ØØd-Ø8   1.Ø87d+ØØ
6.ØØØd-Ø8   1.Ø87d+ØØ
6.1ØØd-Ø8   1.Ø87d+ØØ
6.2ØØd-Ø8   1.Ø87d+ØØ
6.3ØØd-Ø8   1.Ø87d+ØØ
6.4ØØd-Ø8   1.Ø86d+ØØ
6.5ØØd-Ø8   1.Ø86d+ØØ
6.6ØØd-Ø8   1.Ø86d+ØØ
6.7ØØd-Ø8   1.Ø86d+ØØ
6.8ØØd-Ø8   1.Ø86d+ØØ
```

Figure III-31 SPICE: XROM Wordline Selected



```
8.4ØØd-Ø8  1.Ø86d+ØØ
8.5ØØd-Ø8  1.Ø86d+ØØ
8.6ØØd-Ø8  1.Ø86d+ØØ
8.7ØØd-Ø8  1.Ø86d+ØØ
8.8ØØd-Ø8  1.Ø86d+ØØ
8.9ØØd-Ø8  1.Ø86d+ØØ
9.ØØØd-Ø8  1.Ø86d+ØØ
9.1ØØd-Ø8  1.Ø85d+ØØ
9.2ØØd-Ø8  1.Ø83d+ØØ
9.3ØØd-Ø8  1.Ø84d+ØØ
9.4ØØd-Ø8  1.1Ø3d+ØØ
9.5ØØd-Ø8  1.143d+ØØ
9.6ØØd-Ø8  1.2Ø3d+ØØ
9.7ØØd-Ø8  1.292d+ØØ
9.8ØØd-Ø8  1.397d+ØØ
9.9ØØd-Ø8  1.511d+ØØ
1.ØØØd-Ø7  1.629d+ØØ
1.Ø1Ød-Ø7  1.755d+ØØ
1.Ø2Ød-Ø7  1.884d+ØØ
1.Ø3Ød-Ø7  2.Ø19d+ØØ
1.Ø4Ød-Ø7  2.166d+ØØ
1.Ø5Ød-Ø7  2.314d+ØØ
1.Ø6Ød-Ø7  2.461d+ØØ
1.Ø7Ød-Ø7  2.66Ød+ØØ
1.Ø8Ød-Ø7  2.861d+ØØ
1.Ø9Ød-Ø7  3.Ø61d+ØØ
1.1ØØd-Ø7  3.257d+ØØ
1.11Ød-Ø7  3.451d+ØØ
1.12Ød-Ø7  3.645d+ØØ
1.13Ød-Ø7  3.823d+ØØ
1.14Ød-Ø7  3.962d+ØØ
1.15Ød-Ø7  4.1Ø7d+ØØ
1.16Ød-Ø7  4.232d+ØØ
1.17Ød-Ø7  4.336d+ØØ
1.18Ød-Ø7  4.421d+ØØ
1.19Ød-Ø7  4.491d+ØØ
1.2ØØd-Ø7  4.547d+ØØ
1.21Ød-Ø7  4.592d+ØØ
1.22Ød-Ø7  4.629d+ØØ
1.23Ød-Ø7  4.661d+ØØ
1.24Ød-Ø7  4.688d+ØØ
1.25Ød-Ø7  4.711d+ØØ
1.26Ød-Ø7  4.732d+ØØ
1.27Ød-Ø7  4.75Ød+ØØ
1.28Ød-Ø7  4.767d+ØØ
1.29Ød-Ø7  4.782d+ØØ
1.3ØØd-Ø7  4.795d+ØØ
1.31Ød-Ø7  4.8Ø7d+ØØ
1.32Ød-Ø7  4.818d+ØØ
1.33Ød-Ø7  4.829d+ØØ
1.34Ød-Ø7  4.838d+ØØ
1.35Ød-Ø7  4.847d+ØØ
1.36Ød-Ø7  4.855d+ØØ
1.37Ød-Ø7  4.862d+ØØ
1.38Ød-Ø7  4.869d+ØØ
1.39Ød-Ø7  4.875d+ØØ
1.4ØØd-Ø7  4.881d+ØØ
1.41Ød-Ø7  4.886d+ØØ
1.42Ød-Ø7  4.892d+ØØ
1.43Ød-Ø7  4.896d+ØØ
1.44Ød-Ø7  4.9Ø1d+ØØ
1.45Ød-Ø7  4.9Ø5d+ØØ
1.46Ød-Ø7  4.9Ø9d+ØØ
1.47Ød-Ø7  4.913d+ØØ
1.48Ød-Ø7  4.916d+ØØ
1.49Ød-Ø7  4.92Ød+ØØ
1.5ØØd-Ø7  4.923d+ØØ
1.51Ød-Ø7  4.926d+ØØ
1.52Ød-Ø7  4.928d+ØØ
1.53Ød-Ø7  4.931d+ØØ
```

Figure III-32 SPICE: XROM Wordline Unselected



```
1.500d-08   1.480d-05
1.600d-08   1.550d-05
1.700d-08   1.620d-05
1.800d-08   1.655d-05
1.900d-08   1.671d-05
2.000d-08   1.687d-05
2.100d-08   1.662d-05
2.200d-08   1.517d-05
2.300d-08   1.372d-05
2.400d-08   1.226d-05
2.500d-08   7.707d-06
2.600d-08   6.593d-06
2.700d-08   1.187d-05
2.800d-08   1.620d-05
2.900d-08   1.816d-05
3.000d-08   1.875d-05
3.100d-08   1.914d-05
3.200d-08   1.942d-05
3.300d-08   1.970d-05
3.400d-08   2.947d-05
3.500d-08   3.126d-05
3.600d-08   1.605d-05
3.700d-08   4.940d-06
3.800d-08   1.159d-05
3.900d-08   1.824d-05
4.000d-08   1.874d-05
4.100d-08   1.514d-05
4.200d-08   1.220d-05
4.300d-08   9.624d-06
4.400d-08   7.450d-06
4.500d-08   5.276d-06
4.600d-08   5.437d-06
4.700d-08   9.073d-05
4.800d-08   1.760d-04
4.900d-08   2.613d-04
5.000d-08  -2.733d-04
5.100d-08  -1.177d-03
5.200d-08  -2.081d-03
5.300d-08   1.103d-01
5.400d-08   6.091d-01
5.500d-08   1.108d+00
5.600d-08   1.607d+00
5.700d-08   2.369d+00
5.800d-08   3.187d+00
5.900d-08   4.005d+00
6.000d-08   4.553d+00
6.100d-08   4.739d+00
6.200d-08   4.925d+00
6.300d-08   5.104d+00
6.400d-08   5.060d+00
6.500d-08   5.015d+00
6.600d-08   4.970d+00
6.700d-08   4.967d+00
6.800d-08   4.988d+00
6.900d-08   5.009d+00
7.000d-08   5.022d+00
7.100d-08   5.009d+00
7.200d-08   4.997d+00
7.300d-08   4.984d+00
7.400d-08   4.989d+00
7.500d-08   4.998d+00
7.600d-08   5.007d+00
7.700d-08   5.010d+00
7.800d-08   5.003d+00
7.900d-08   4.997d+00
8.000d-08   4.991d+00
```

Figure III-33 SPICE: XROM Output Going High

```
9.400d-08   5.032d+00
9.500d-08   4.963d+00
9.600d-08   4.730d+00
9.700d-08   4.033d+00
9.800d-08   2.768d+00
9.900d-08   1.587d+00
1.000d-07   5.832d-01
1.010d-07   2.196d-01
1.020d-07   5.715d-02
1.030d-07  -3.575d-02
1.040d-07  -1.890d-02
1.050d-07  -2.042d-03
1.060d-07   1.481d-02
1.070d-07   9.068d-03
1.080d-07   3.047d-03
1.090d-07  -2.975d-03
1.100d-07  -3.432d-03
1.110d-07  -1.517d-03
1.120d-C7   3.973d-04
1.130d-07   1.376d-03
1.140d-C7   1.822d-04
1.150d-07   9.638d-05
1.160d-07   4.989d-05
1.170d-07   1.897d-05
1.180d-07   2.202d-05
1.190d-07   1.933d-05
1.200d-07   1.802d-05
1.210d-07   1.674d-05
1.220d-07   1.773d-05
1.230d-07   1.941d-05
1.240d-07   2.508d-05
1.250d-07   4.738d-05
1.260d-07   9.897d-05
1.270d-07   1.297d-04
1.280d-07   1.367d-04
1.290d-07   1.280d-04
1.300d-07   1.086d-04
1.310d-07   9.090d-05
1.320d-07   6.924d-05
1.330d-07   3.620d-05
1.340d-07  -2.888d-06
1.350d-07  -3.948d-05
1.360d-07  -1.165d-04
1.370d-07  -2.109d-04
1.380d-07  -2.748d-04
1.390d-07  -2.995d-04
1.400d-07  -2.553d-04
1.410d-07  -2.112d-04
1.420d-07  -1.476d-04
1.430d-07  -6.711d-05
1.440d-07   1.342d-05
1.450d-07   8.665d-05
1.460d-07   1.122d-04
1.470d-07   1.378d-04
1.480d-07   1.633d-04
1.490d-07   1.414d-04
1.500d-07   1.022d-04
1.510d-07   6.298d-05
1.520d-07   3.465d-05
1.530d-07   2.815d-05
1.540d-07   2.165d-05
1.550d-07   1.515d-05
1.560d-07   2.519d-05
1.570d-07   3.643d-05
1.580d-07   4.767d-05
1.590d-07   4.918d-05
1.600d-07   4.214d-05
```

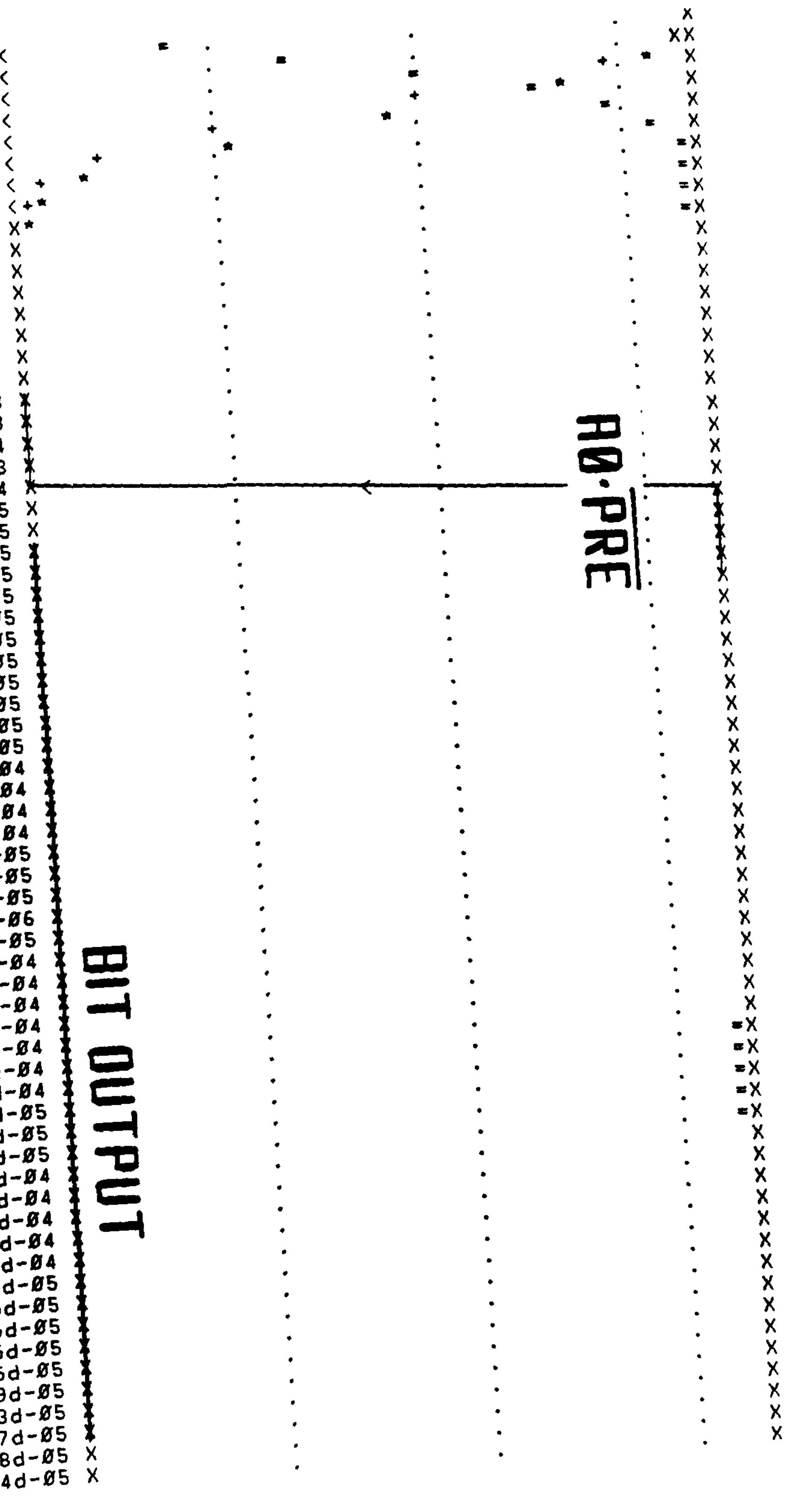


Figure III-34 SPICE: XROM Output Going Low

IV. Automatic Generation of the XROM

Overview

The XROM circuit used on the WFTA16 must store 4 1/2K of 12-bit addresses to calculate the 16, 240, 272, and 4080 point DFTs. The WFTA15 and WFTA17 chips must store a similar number. Thus, an efficient procedure to personalize the 54K bits within the XROM must be found. Obviously, manual personalization of the XROM using Caesar would be very time consuming and error prone. The placement of the personalization ones and zeros is further complicated by the XROM's intricate design. Therefore, software that automatically produces a layout description of the personalized XROM for a given list of data addresses is essential.

Since a computer program is to be developed to personalize the XROM, it would be desirable to write the program to optimize certain parameters within the XROM before generating the layout. The software will therefore, include programs that attempt to optimize the power dissipation, speed, and yield of the XROM by minimizing the number of transistors and drains in the XROM array.

This chapter will present the development and implementation of the software that optimizes the bit pattern arrangement of the XROM personalization and generates a Caesar file layout description of the result. It will be shown that the transistor minimization problem can be cast as the classical graph partitioning problem, and that the drain removal problem is a disguised version of the Traveling Salesman Problem. The information in this chapter is presented in the following order. First, the development and procedure used to minimize the total number of devices in the XROM is described. Second, the development and procedure used to minimize the total number of drains in the XROM is described. Third, the software that implements the two optimization routines is presented. Finally, the methods and software used to automatically generate the XROM personalization are presented.

XROM Optimization: Minimizing Total Devices

Overview. If the total number of devices or "ones" in the XROM can be significantly reduced, the yield, power dissipation, and possibly the speed of the XROM will be improved. Since the percent yield of any CMOS LSI circuit is inversely proportional to the number of active devices per square micron of silicon (Ong, 1984:343), deceasing the number of devices in the XROM will result in improved yield. The speed of operation for the XROM will generally be improved by removing devices since worst-case wordline gate capacitances and bitline drain capacitances will usually decrease. As a result of the decrease in the overall XROM capacitances described above, the switching power dissipation will decrease proportionally.

The remainder of this section will describe the manner in which the total number of devices in the XROM are minimized. Basically the procedure consists of varying the address lines used for particular address positions, and applying a number of complex sign bit operations to each addressing configuration until the minimum number of devices is achieved.

<u>Varying the XROM's Addressing Scheme</u>. The number of address lines used for a particular XROM depends on its storage size. The swapping of two or more addresses to the XROM will only change the placement position of each bit in the array, not the total number of one-bits. What makes the addressing scheme important in minimizing the devices is the fact that the changed bit placements may increase the usefulness of the XROM sign bits. That is, certain bit placements may have a "better" clustering of ones than others, and these clusters of ones can be eliminated by the correct placement of an inverting sign bit. The organization of the XROM's sign bits will be presented in the next section. However, this organization is such that only the address lines used for the multiplexer at the top of the XROM bitline columns and the address that is input directly to the XROM array can cause the sign bits to be more or less effective. Changing the address line order in the PLA

wordline decoder will only change the location of data and sign bits, not their 0 or 1 state.

Additionally, certain addresses may not be used to try to improve the sign bit effectiveness. This limitation applies to any higher ordered address that does not run the complete binary number range due to the XROM being a size other than a power of two. For example, the XROM for the WFTA16 chip has 4 1/2K of data words. Since the XROM outputs four words at a time, 1 1/8K addresses are needed to access all the contents of the XROM. The higher ordered four address positions of the 11 bits required do not cycle through to the maximum addressing capability of 2K. Thus, these four addresses must always be used in the PLA wordline decoder and cannot enter into the optimization routine. If these addresses were considered, and were input directly to the XROM array or the multiplexer, the addresses applied to the XROM would not run sequentially. A nonsequential input to the XROM is not desirable for this application and will not be considered due to the additional complexity it would introduce.

Therefore, the use of the address lines to minimize the number of devices in the XROM shall proceed in the following manner:

1. The bits of the XROM are placed for each possible address configuration that can affect the sign bit effectiveness (for the WFTA16 XROM there are 7 candidates for the address column and 6!/4!2! combinations for the multiplexer addresses = 105 configurations).

2. The sign bit scheme described below is applied for each address scheme.

3. The address and sign bit scheme that produces a minimum number of devices is used.

<u>XROM sign bits</u>. If the XROM (or any other ROM) used no sign bits and did not invert the state of any row or column, the maximum number of possible transistors in the XROM would be equal to the size of the XROM. For example, the XROM used on the WFTA16 chip could be personalized with up to 54K ones in the worst case. Of course since the actual contents of that XROM are sequential addresses, the number of ones is more near half that figure. Figure IV-1 shows the distribution of the number of transistors in a 54K XROM for six classes of stored data. The six transistor distributions of Figure IV-1 are for 50 randomly generated data cases; WFTA15, WFTA16, and WFTA17 addresses; and 4080-point WFTA and DFT coefficients. For each type of data, the number of cases that resulted in a particular number range of transistors per XROM are given. Note that the figure depicts the 50 random distributions with an average number of ones equal to 27,632 with a standard deviation of 103. The 54K WFTA15, WFTA16, and WFTA17 XROMs storing addresses have an average number of ones of 26,388, 26,448, and 26,504 respectively. The four 54K XROMs storing transform coefficients average 27,577 transistors.

In any of the above situations, the average number of

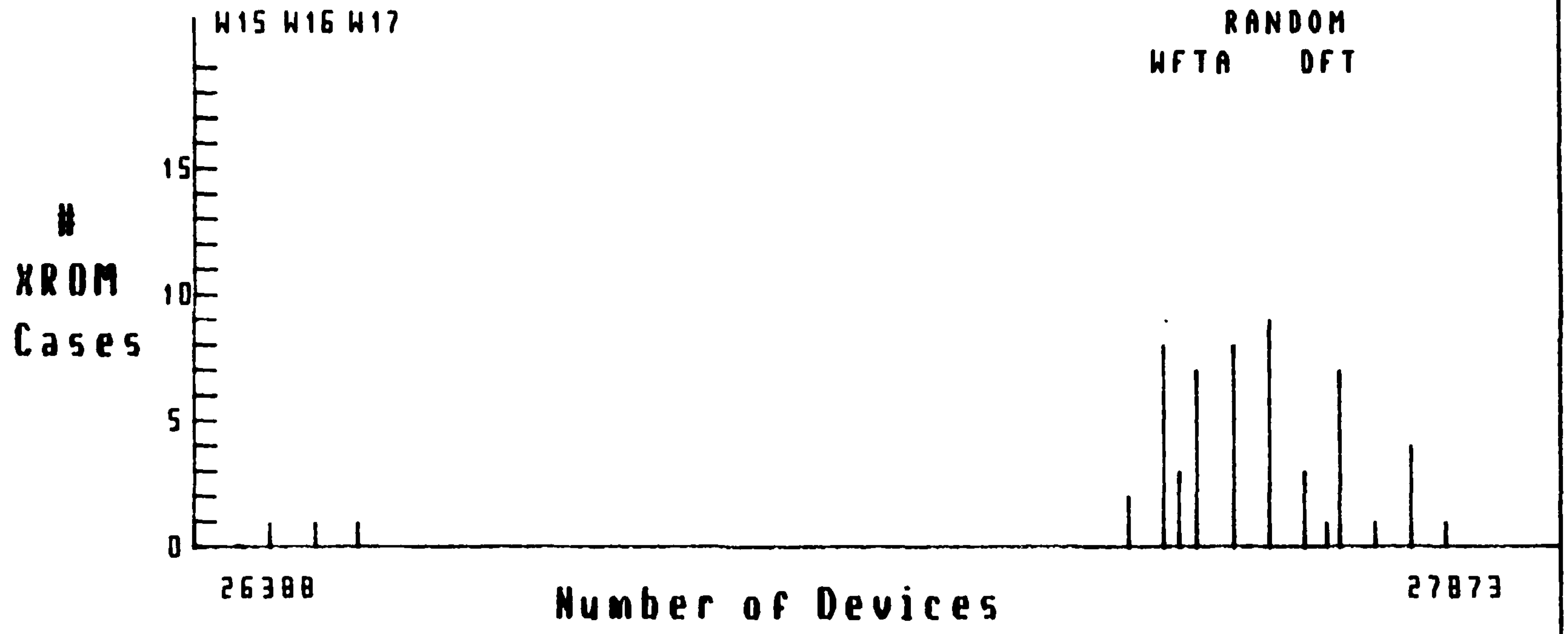


Figure IV-1 Distribution of Ones:
No Sign Bits

ones in an XROM may be reduced if a single sign bit is used for the entire XROM array. The sign bit is set and all bits in the XROM are inverted if the original number of one-bits is over half the XROM size. The sign bit is not set and the XROM is left as is if the number of one-bits is less than or equal to half the XROM's size. If the sign bit is set, the data is inverted (again) when output to restore its original sense. Situations where the total number of ones is less than or equal to half the XROM size will not produce a decrease in the number of total ones using this method. However, on the average a single sign bit will decrease the number of ones in the array. Figure IV-2 shows the same six XROM data class cases as Figure IV-1 only a single sign bit has been applied to the entire XROM. Notice that the random case distribution is no longer normal and the average number of ones has shifted down to 27,561. The results for the other five classes of data show little or no gain in the number of transistors.

The use of sign bits can be extended to one sign bit for each half, quarter, eighth, etc., of the XROM array. Each step adding some complexity, but producing a smaller number of ones in the array on average. This increase in sign bit effectiveness is due to the fact that if a sign bit governs a "small" number of bits, the overall odds that the number of ones will not be near half the size of the group of bits are better than the odds would be if the sign bit governed

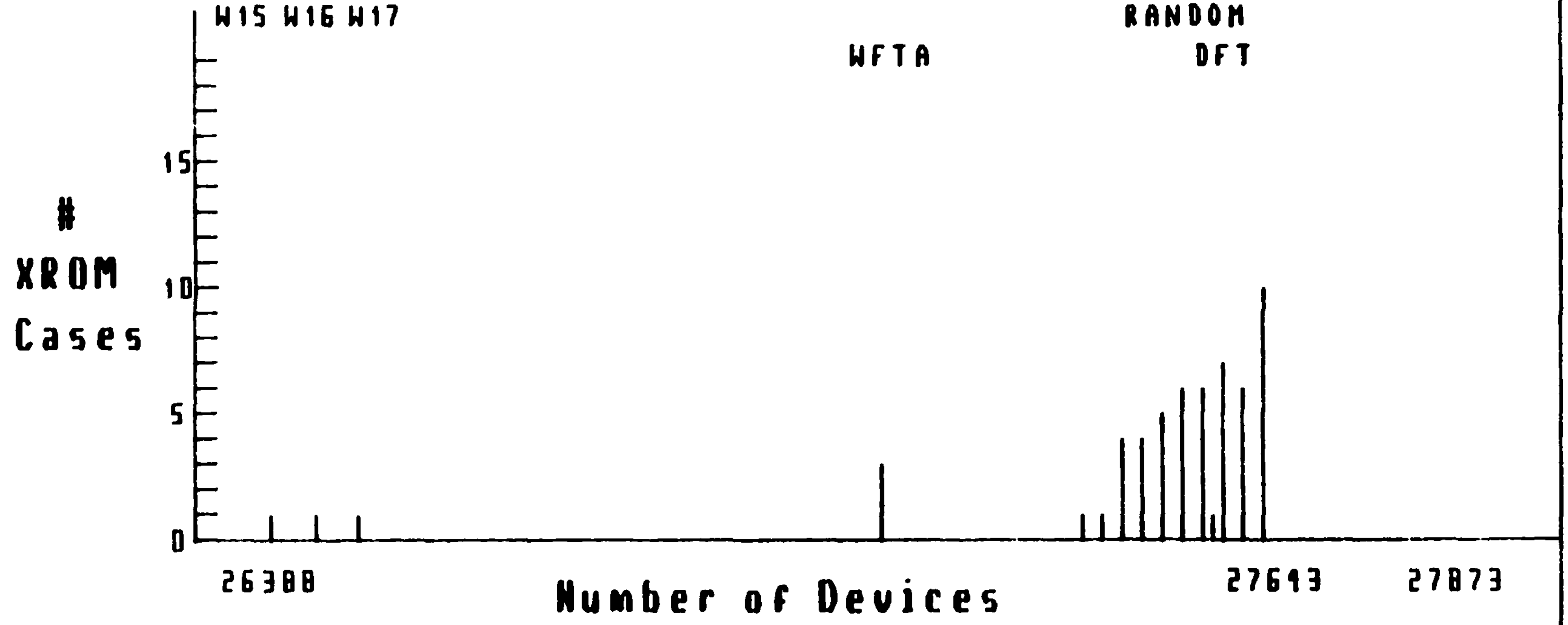


Figure IV-2 Distribution of Figure IV-1:
Single Sign Bit

a "large" group of bits. For example, if the division process is taken to the extreme, one sign bit could govern one XROM bit, and an XROM array with all zeros would be guaranteed. Unfortunately, the contents of the original XROM would be duplicated in the sign bit array that is as large as the XROM.

The use of one sign bit for each data word (be it a random number, address, transform coefficient, or two's complement number) seems intuitively appealing since it would have the sign bit governing a small number of bits, have a straightforward sign bit assignment procedure, and there is some "correlation" at the word level (for example, negative two's complement numbers with a small magnitude). However, since the XROM for the WFTA chips has a four to one multiplexer for each output data bit, the sign bits would have to be multiplexed also. This would add significantly to the access time of the XROM. Additionally, one sign bit for each data word would have a significant layout area cost. Thus, a single sign bit (to be referred to as a word sign bit) for each of four data words in a group was used on the WFTA chip XROMs. Figure IV-3 shows the six XROM cases used in Figure IV-1 with a sign bit applied for every four data words. All six types of data stored in the XROM show a substantial decrease in the average number of transistors per XROM over the results given in Figure IV-1.

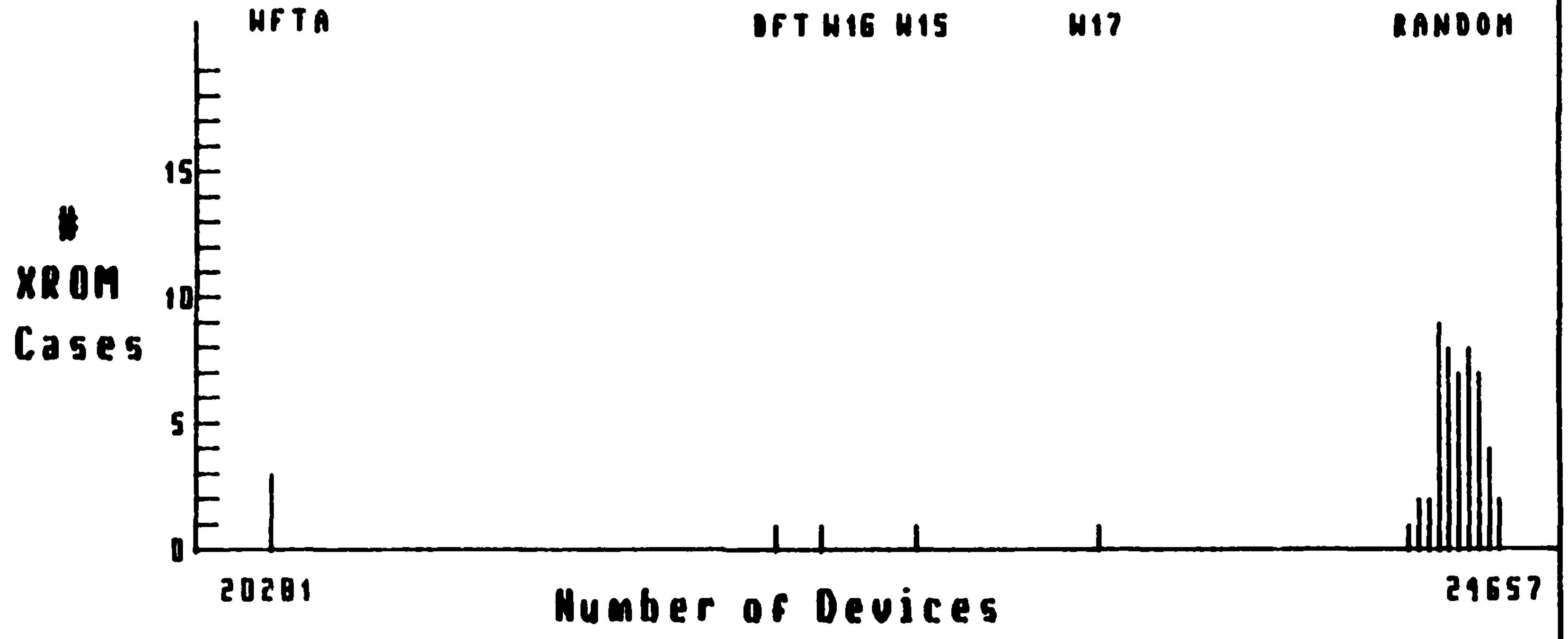


Figure IV-3 Distribution of Figure IV-1:
Sign Bit for Every Four Data Words

One final "sign bit" operation was used on the XROM. Since the word sign bit contents must be used by each column's sense amplifier to determine if the bit or inverted bit sense is to be output, the bits in an entire column of the XROM can be inverted by using a sense amplifier with the word and word' lines reversed. The use of this inverting column sense amplifier will be referred to as a "column sign bit." It is important to note that because each bitline into the XROM's 4 to 1 multiplexer can be the A0 or A0N bit, and because one sense amplifier operates on the output of the multiplexer, a column sign bit governs a column that is eight bits in width (a column-byte). Figure IV-4 shows the six XROM cases used in Figure IV-1 with the column sign bits and then the word sign bits applied.

At this point, the combination of varying the addressing scheme of the XROM and the application of column and word sign bits produces a distribution of ones for the six cases as shown in Figure IV-5. During this phase, word sign bits were applied both before and after the column sign bits. The XROM solution with fewer transistors was used.

Prearranging XROM Columns. The effectiveness of the XROM's word sign bits can be improved by rearranging the column-bytes before the word sign bits are applied. If column-bytes that are similar (or strongly correlated as determined by a correlation scheme) are grouped together

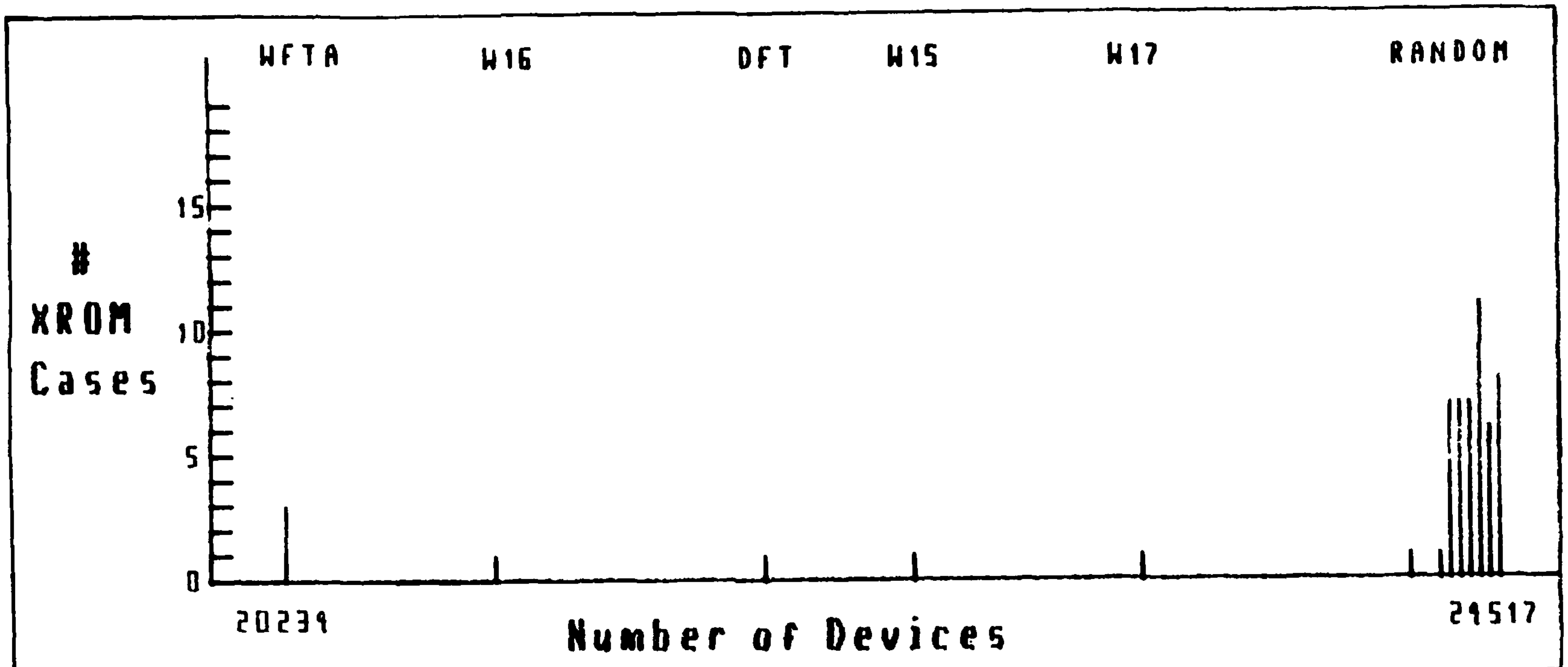


Figure IV-4 Distribution of Figure IV-1: Column and Word Sign Bits

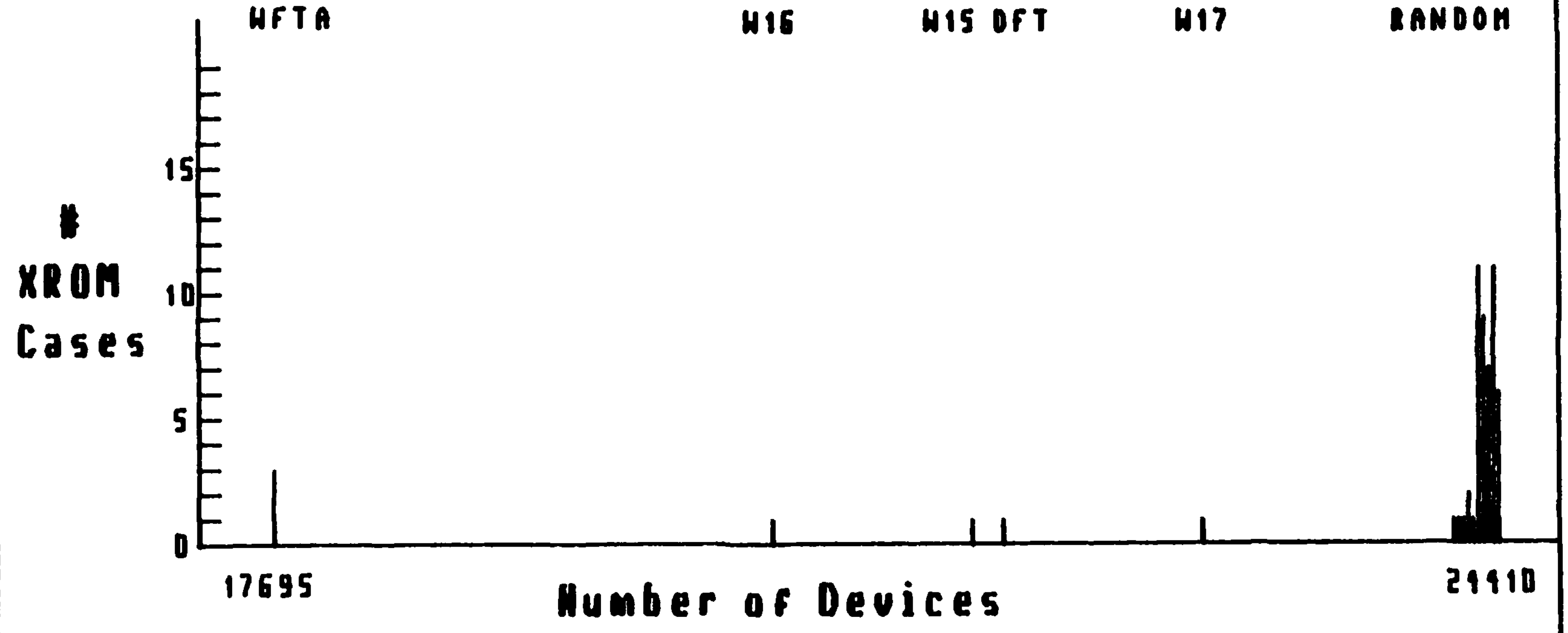


Figure IV-5 Distribution of Figure IV-4: Varying Addresses

under the control of a single word sign bit column, a lower total number of devices (or ones) in the XROM can be achieved. Thus, the method to obtain a minimum number of devices develops into the following procedure.

1. Place the bits for each of the particular address schemes.
2. Save the placement pattern for step 8.
3. Apply column sign bits.
4. Calculate the column correlation distances as determined by the correlation scheme.
5. Group the columns with maximum correlation distances between them together.
6. Apply the word sign bits.
7. Note result.
8. Restore the current address scheme's original bit placement, and apply steps 4, 5, 6, and 3 in that order.
9. Note result.
10. Return to 1. unless the possible address schemes have been exhausted.
11. Use the result that yields a minimum number of XROM devices.

The column arrangement procedure yields a lower number of devices by grouping column-bytes that have the most similar one/zero patterns under the control of a single word sign bit column regardless of what data word that column-byte belongs to. However, the proper data words must be recovered from the scrambled column-bytes. The two items of this procedure that are yet to be described are the correlation scheme used, and the method of partitioning the strongly correlated columns into groups.

The Column Correlation Scheme. A simple and efficient technique is needed to determine the number of ones that can be removed from the XROM array if two given column-bytes are placed under the control of the same word sign bit column. Since word sign bits exist on each row of the XROM, the problem can be reduced to finding the correlation distance between two eight-bit bytes of data and applying that measure to all the rows of the two column-bytes of interest. The measure for each row can then be summed over all rows for the total correlation distance between two column-bytes.

Since each eight-bit byte in a column (and each word sign bit in a word sign bit column) has a column address (A0) and a column address' (A0N) portion, the correlation scheme must total the similarities between column-bytes for the A0 half and the A0N half. Table IV-1 shows the correlation value matrix used to calculate the correlation distances between two bytes within a column. The correlation value matrix assigns a complex number to each data byte in the XROM array. The Manhattan magnitude of this complex number indicates the byte's potential for zero bits (over four). Thus, a byte with half ones and half zeros in both A0 and A0N nibbles receives a correlation value of 0. It has no potential to obtain over four zeros in that byte whether

NUMBER OF ONES IN REAL (A0) HALF OF BYTE

4	2 - 2J	2 - J	2	2 + J	2 + 2J
3	1 - 2J	1 - J	1	1 + J	1 + 2J
2	- 2J	- J	0	J	2J
1	-1 - 2J	-1 - J	-1	-1 + J	-1 + 2J
0	-2 -2J	-2 - J	-2	-2 + J	-2 + 2J
	0	1	2	3	4

NUMBER OF ONES IN IMAGINARY (A0N) HALF OF BYTE

Table IV-1 Correlation Value Matrix

it is inverted or not. High potentials (2-2j, 2+2j, -2-2j, -2+2j) are awarded to bytes that have an A0 half with four ones (+2) or four zeros (-2) and A0N halfs with four ones (+2j) or four zeros (-2j). The bytes with the high correlation values shown above can end up with all zero bits in the byte since 2 sign bits operate on each byte (A0 and A0N).

The importance of the correlation procedure is that the A0 and A0N word sign bit pair must invert or not invert the bits under its control based on the total bit count for all data bytes in its data word area. For the WFTA chip's XROM, there are 12 data bytes in each data word. If column-bytes with all ones are grouped with an equal number of column bytes with all zeros, there is no potential to obtain any more than one half zeros in that data word area even though individual correlation values are high. Therefore, the correlation values must be used to calculate the correlation distance between every column-byte pair in the XROM, and a partitioning algorithm must be later applied to group those most correlated together. The correlation distance between column-byte x and column-byte y is calculated by,

$$\sum_{i=0}^{Rows-1} \left| Re(x_i + y_i) \right| + \left| Im(x_i + y_i) \right| .$$

As indicated previously, the same correlation metric is applied over all rows and summed. Each row calculation determines the real (A0) potential for zeros (over two) and the imaginary (A0N) potential for zeros (over two) for two bytes and adds them.

The result of applying this correlation distance metric to all column-byte pairs of the XROM is a symmetric, fully connected graph with each column-byte serving as a vertex.

<u>Solving the Graph Partitioning Problem</u> (Kernighan and Lin, 1970). For the WFTA chip XROMs, the column-bytes must be partitioned into groups of 12. Each column-byte has a symmetric edge distance to each of the remaining 47 column-bytes in the XROM. The column-bytes can be viewed as the vertices of a symmetric digraph with $((48)^2/2)-(48/2) = 1128$

edges. This section will present an efficient heuristic procedure to partition this graph into the four groups of 12 so as to minimize the total cost of the edges cut (or maximize the correlation cost within the group). The procedure was developed by B. W. Kernighan and S. Lin in 1969.

The graph partitioning problem is in a class of very hard problems known as NP-complete (Hyafil and Rivest:1973). An exhaustive search for the optimum solution to the problem at hand would result in

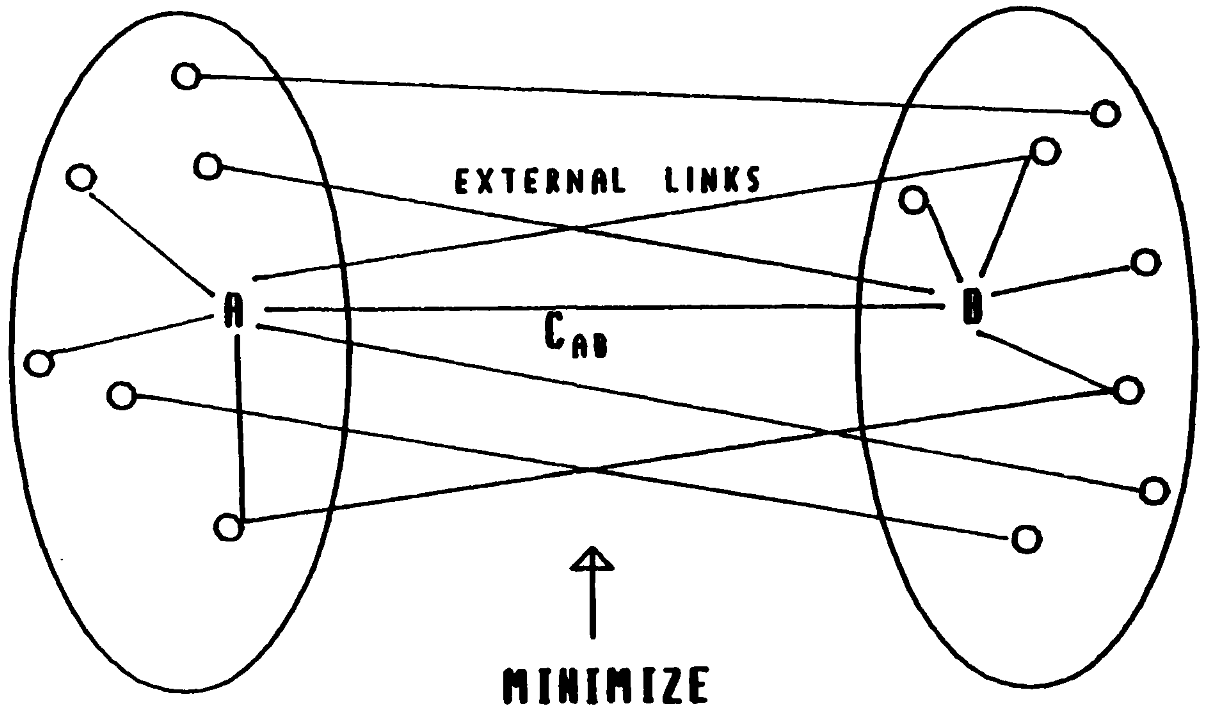
$$1/4! \times \binom{48}{12} \times \binom{36}{12} \times \binom{24}{12} \times \binom{12}{12} = 2.7 \times 10^{50} \text{ cases.}$$

Thus, we must turn to a heuristic approach.

The Kernighan and Lin heuristic divides a given graph of 2n vertices into two subsets of n vertices each. Thus, the heuristic must be applied again to each partition of 24 column-bytes after they are generated. The heuristic starts with any arbitrary partition A, B of set S, and tries to decrease the external cost of the partitions by a series of interchanges of subsets of A and B. The external cost is the sum total of all link costs between each vertex in one partition and all the vertices in the other partition. When no further improvement is possible from interchanging subsets of A and B, a local (perhaps global) minimum has been found. Repeated application of the algorithm on arbitrary starting partitions provides a fairly high probability of obtaining the optimum partition (75% probability in this application and a 100% probability that the solution is within 10% of the optimal).

The algorithm's power results from the use of a difference measure, D, for each node in the graph (see Figure IV-6). The D value is the difference between a node's total external link costs and its internal link costs (sum of all link costs to nodes within the same partition). Any node with a high D value is a good candidate for exchange. Although the two nodes with maximum D values in each partition will not always give the greatest gain, examining the highest three D values in each partition will produce the greatest gain in virtually all cases.

The node exchanges during one pass continue until all the nodes of partition A are in B and vice versa. At this point, the sequence of gains achieved during the process of totally swapping all nodes is examined. If a positive gain value is observed at any stage, the maximum positive gain will determine how far to actually proceed in the swapping. The nodes are swapped to that point, D values are recalculated, and the process continues until no positive gain is found. By applying this heuristic to the 48 column-bytes and then to each 24 column-byte group, a 4-way partition of the 48 column-bytes into maximally correlated groups of 12 is achieved.



$D_I = \sum$ External links $- \sum$ Internal links

Gain $= D_A + D_B - 2C_{AB}$

Figure IV-6 Kernighan and Lin Graph Partitioning

It is worth mentioning at this point that an XROM test case was contrived that showed a reduction from 27K ones down to zero ones after including the column swapping step. The case had an alternating pattern of ones and zeros in the XROM array which could not have any ones removed using all previously discussed techniques except the column correlation and partitioning. By invoking the column swapping step in the procedure, 54K zeros (or zero devices) was the final result.

Revisiting the six XROM cases presented in the "XROM sign bit" section, and applying all device minimizing techniques described, results in the number of ones distribution shown in Figure IV-7.

Table IV-2 summarizes the number of devices obtained for each of the various stages of techniques presented in the device minimization procedure on the six XROM test cases. The results of Table IV-2 are shown graphically in Figure IV-8. Notice that the decrease in number of XROM transistors is significant in all cases with the possible exception of the random data. The WFTA16 XROM exhibits a 40% decrease in transistors by applying all of the device minimization techniques.

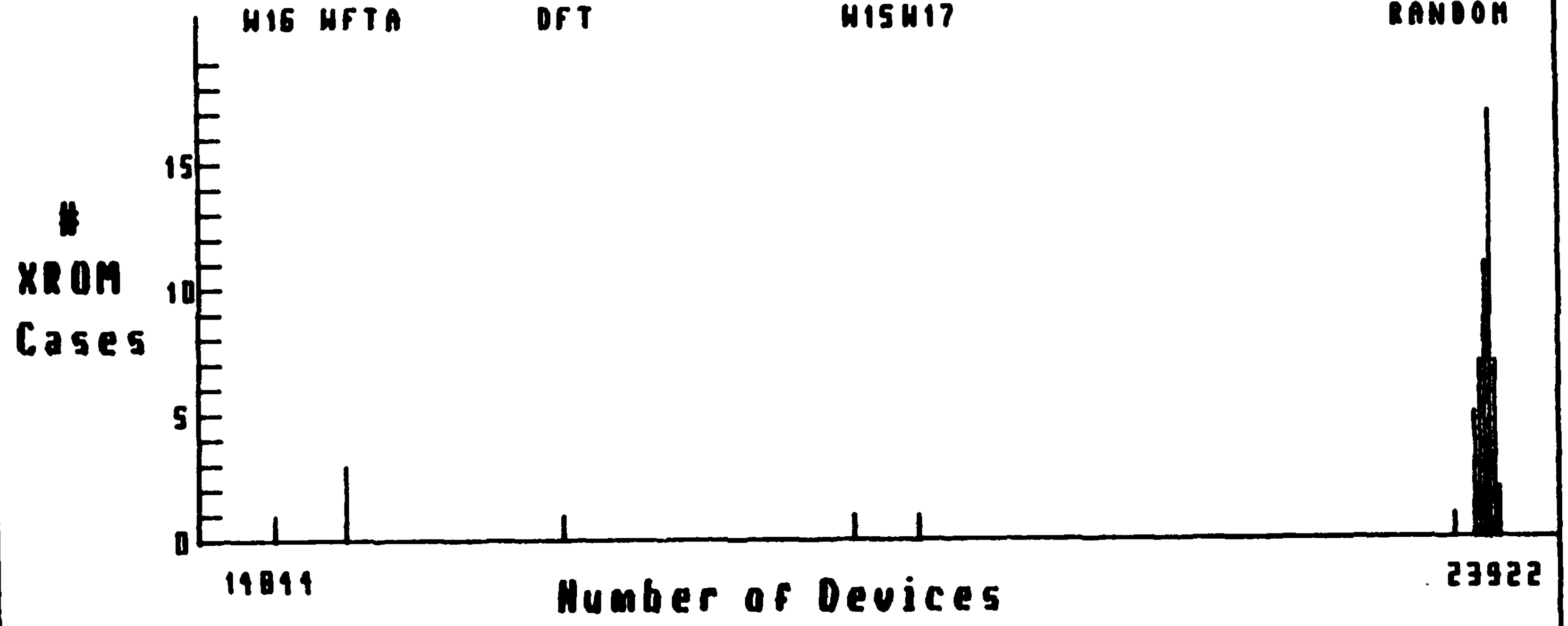


Figure IV-7 Distribution of Figure IV-1:
All Device Minimization Techniques Applied

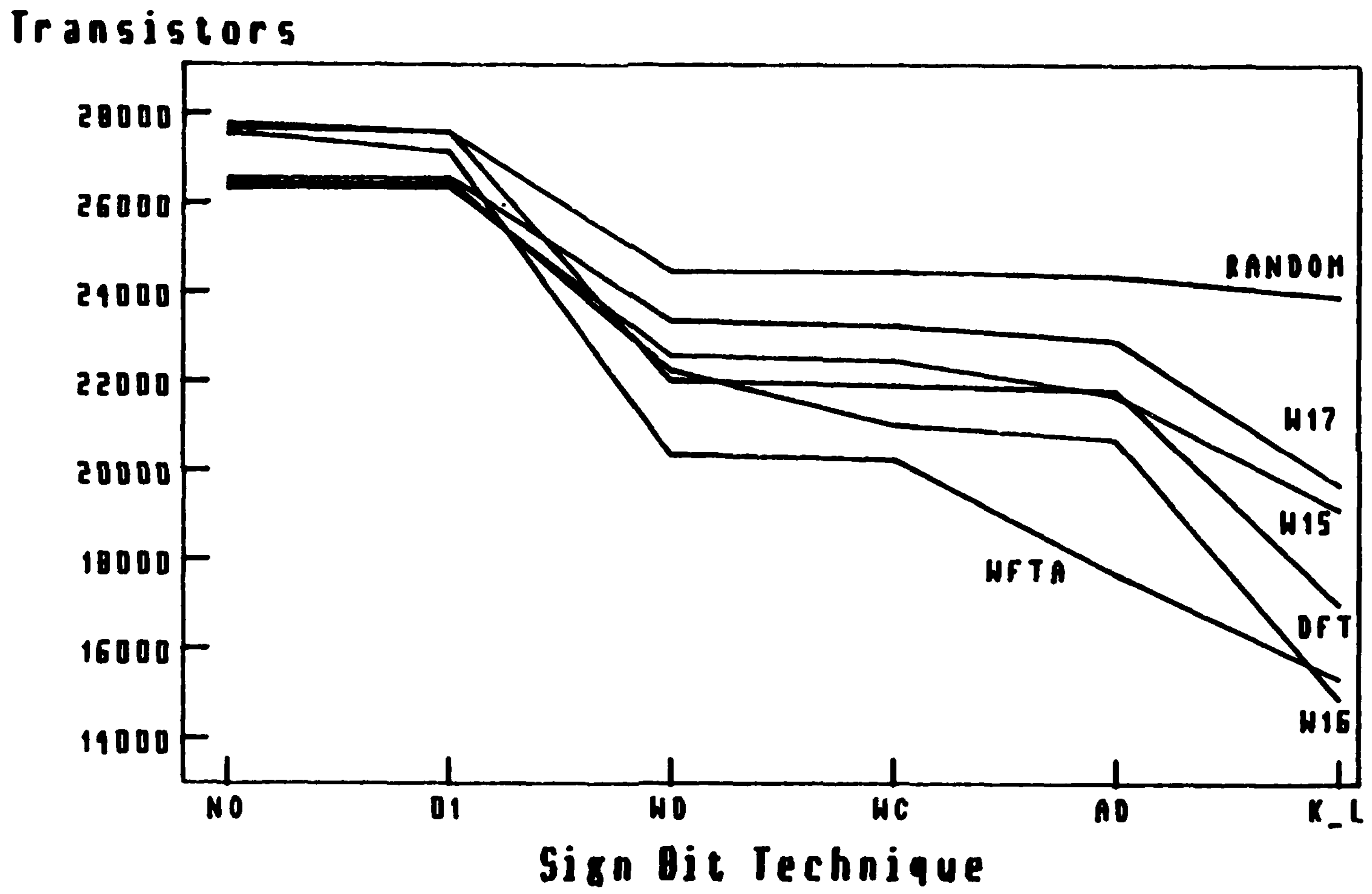


Figure IV-8 Device Minimization Results

CASE / SIGN BITS	W15	W16	W17	DFT	WFTA	Random
NO	26388	26448	26504	27725	27528	27632
01	26388	26448	26504	27571	27152	27561
WD	22546	22194	23292	22035	20281	24478
W/C	22443	20964	23254	21913	20234	24423
AD	21667	20624	22862	21819	17695	24327
K & L	19138	14844	19617	16998	15387	23858

Devices

Table IV-2 XROM Device Minimization Summary.

XROM Optimization: Minimizing Total Drains

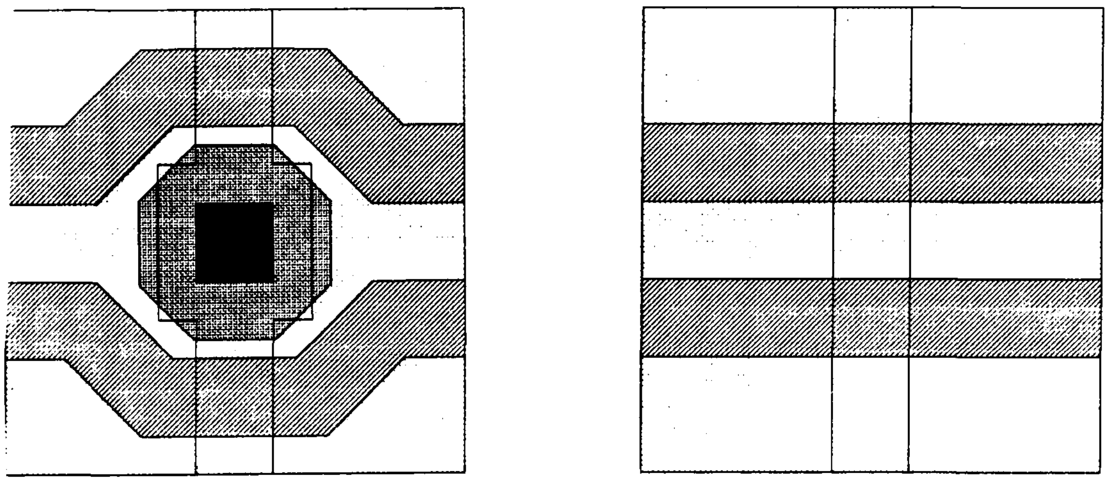
Overview. Many bit positions of an XROM will have zero personalizations. As was shown in the previous chapter, a zero personalization is accomplished by removing or not placing a device between the appropriate column address (A0 or A0N) drain and the bitline drain. If a column address line drain or a bitline drain has none of the four possible devices (ones) connected to it, it serves no purpose other than to slow the operation of the XROM, increase power consumption, and decrease yield. Two possible cell alterations can be performed on a cell whose drain does not have any active devices. They are 1) removing the unused drain diffusion and metal contact, and 2) straightening the polysilicon wordlines that will no longer need to be routed around the drain. The use of these two reconfigurations can improve the overall speed, power dissipation, and yield of the XROM. A graphic representation of the improved cell is shown in Figure IV-9.

The improvement in speed is a result of the decrease in wordline length and the decrease in bitline and address column line capacitance. The shorter wordline will result in a lower resistance, thereby decreasing the time needed to raise or lower the voltage on the wordline. The decrease in drain capacitance on the metal column lines will allow them to switch faster.

The power savings result from the decrease in the column line capacitance. Since power is determined by

$$P = CV^2f,$$

a decrease in capacitance results in a corresponding decrease in power (Weste and Eshraghian, 1985:148).



(a) standard cell (b) no drain cell

Figure IV-9 Reconfiguring XROM Cells With No Drain.

The possibility for an improvement in yield results from the fewer diffusion implants that must be correctly placed in the XROM array. With fewer implants, the probability of an error on the chip decreases. This improved yield becomes especially significant if the XROM array design is used for a one megabit ROM chip.

The remainder of this section will describe the manner in which the total number of drains in the XROM are minimized. The procedure is one of rearranging columns and rows in order to group four zero personalizations around the largest number of drains possible. Then each cell whose drain is surrounded by four zeros is replaced by the no drain cell of Figure IV-9. The complexity of the procedure is rooted in trying to find the optimal ordering of rows and columns without exhaustively searching all possibilities. It will be shown that this problem, like the graph partitioning problem, is NP-complete.

Reordering Columns and Rows. The objective is to remove as many drains as possible from the XROM array. A great deal of progress toward reaching that objective is achieved by minimizing the number of devices in the XROM. For example, if a 54K XROM contains 27K ones, the probability of a drain being pulled is 1/16th for a random ordering of rows and columns. This probability is calculated by multiplying the probability of a zero being at a given bit position (probability = 1/2) times the same probability at each of the three other bit positions around a single drain. Thus, on the average 1/16 times the total number of drain positions in the 54K XROM will be removed. If the device minimization procedure (described in the preceding section) is applied to the XROM and decreases the number of ones in the XROM to 13.5K, the probability of a drain being pulled increases to almost 1/3rd (3/4 to the fourth power). Thus, the average total number of drains removed from the 54K XROM increases to 1/3rd times the total drain positions. This is over a five-fold increase in drains pulled for a two-fold increase in devices removed.

As in the preceding section, six classes of XROM data will be used to demonstrate the increase in drain removal achieved by the device minimization routine. Figure IV-10 shows the distribution of drains remaining for the six data types in a 54K XROM before any optimization steps (this distribution contains 15 random samples rather than 50). The average percentage of drains pulled is 6.94 percent for the random cases and 24.71 percent for the WFTA16 addresses. Figure IV-11 shows the distribution of drains remaining for the same six cases of Figure IV-10 with the device minimization procedure applied. The average percentage of drains pulled is 11.21 percent for the random cases, and 44.01 percent for the WFTA16 addresses.

Thus, a large number of drains can be removed without attempting to optimize the layout specifically to pull drains. However, with some extra computing time and a minimal increase in complexity, more drains can be removed by reordering the columns and rows of the XROM to obtain groups of four zeros around drains. The only restriction on the reordering procedure is that it does not affect the device minimization results by moving column-bytes away from its controlling sign bit column. This means that when columns are rearranged, they may only be swapped within the data word group that is controlled by a word sign bit column pair. Rows, on the other hand may be reordered in any manner since the word sign bits will be moved with the row bits that they govern, and the NAND PLA decoder allows any wordline row personalization required.

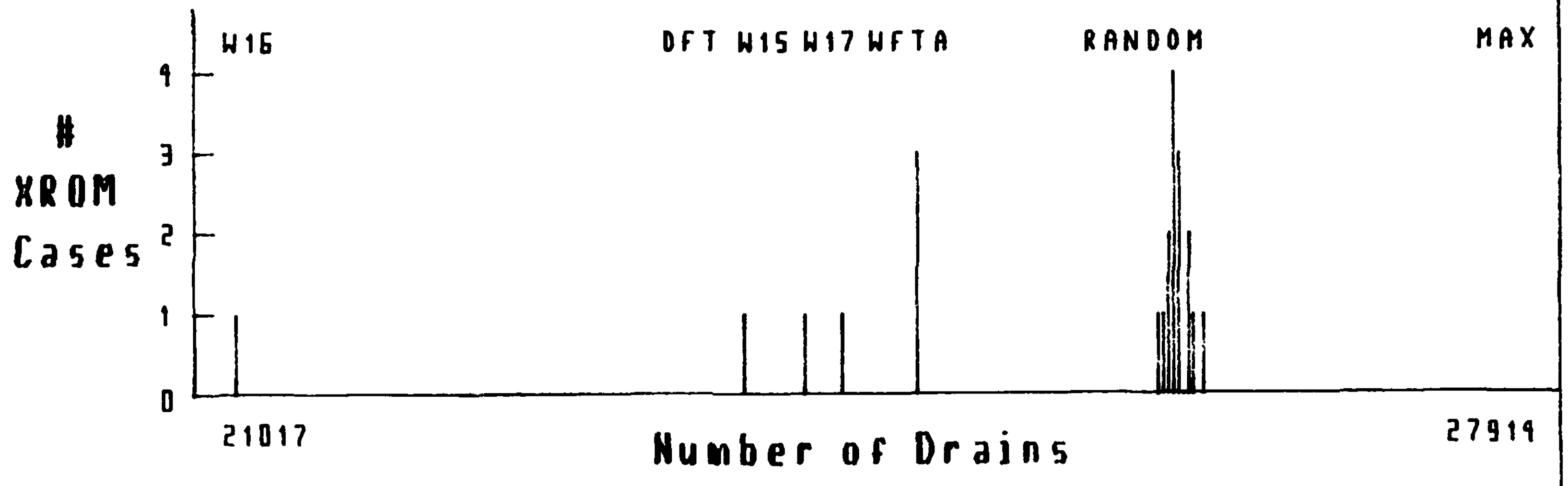


Figure IV-10 Distribution of Drains: Non-optimized

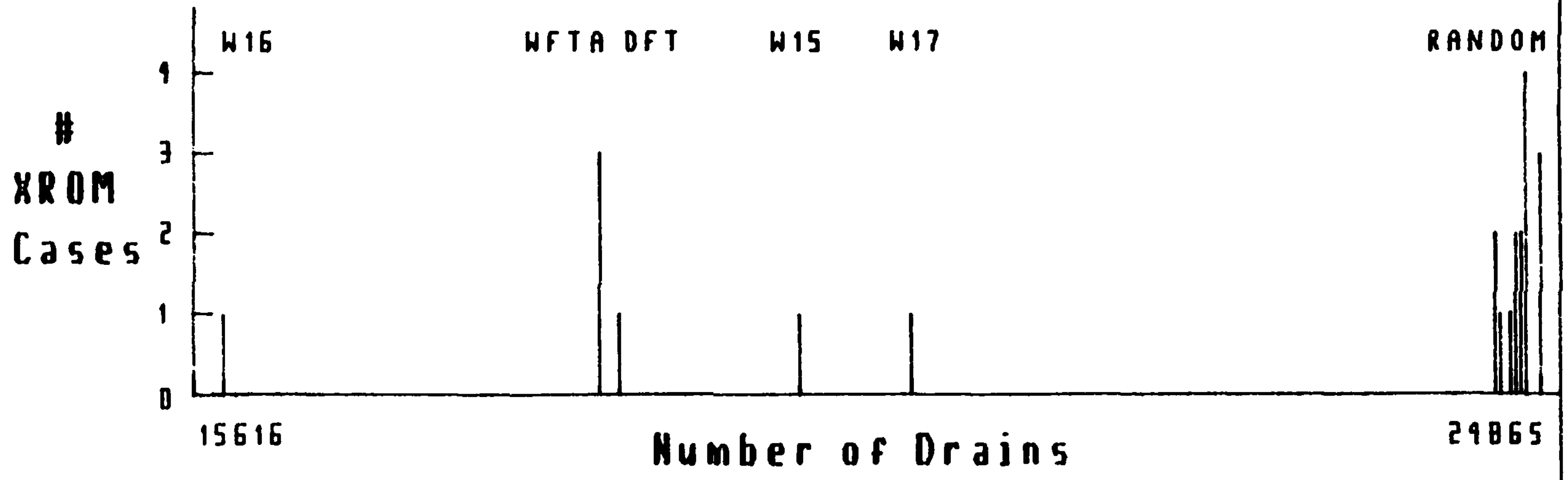


Figure IV-II Distribution of Drains: Device Minimized

Thus, in order to minimize the number of drains in the [illegible]TA16 XROM array, data bit columns within each group of 12 will be permuted as will all 144 rows of the XROM. To exhaustively search all possible row and column combinations to find the minimum number of drains would be prohibitive. There are 144! possible ways to place the rows. For each placement, there are four groups of column-bytes each with 12! possible column-byte orderings. Additionally the columns within a column-byte can be ordered 4 different ways. Thus, there exist 144! x 4 x (12! x 4^{12}) possible unique orderings that could be examined to find the one with the greatest potential for drain removal.

Obviously, an exhaustive search will not be possible, and a heuristic solution must be developed. As is often the approach with large, intractable problems, a division of the problem into two or more subproblems will be performed. Although, this approach (and others to follow) will not always yield the optimum solution, it does allow a solution (and a reasonably good one) to be found.

Since the columns of the XROM intermesh with the rows, a heuristic which first groups zeros about drains for columns and then for rows would not be very effective. This is because many of the groups of four zeros about drains produced by the column ordering would be broken up when the row ordering attempts to find a better solution. Certain row exchanges may have to be disallowed in order to keep specific desirable results from the column ordering. The heuristic will not take two steps toward solving the problem. Rather, it will take one partial step toward a solution and then make the next step dependent on the last.

The heuristic cannot use a physical division of the XROM either. Since the entire row or column must be switched, a division of the layout into segments and grouping zeros in that piece of the XROM is not a workable solution.

If the columns are ordered to maximize the zero pairs along the wordlines in the XROM, the ordering of the rows can attempt to maximize the grouping of paired zero-pairs to remove drains. In this way, a heuristic which uses a division of the original problem into two independent sequential problems is obtained. The heuristic seeks to maximize the basic element of a group of four zeros before attempting to maximize the groups of four themselves.

However, another problem has spawned from the solution of the first. Another heuristic is needed to group zero pairs in the column ordering and to group four zeros about a drain for the rows ordering. If all possible column pairings were attempted and then all row pairings to achieve the solution, the number of cases required to be examined would not be much less than the original exhaustive search already examined. The number of unique orderings that would have to be examined with this heuristic is

$$144! + 4 \times (12! \times 4^{12}).$$

Therefore, a smarter way to order the 144 rows and 12 column-bytes per group must be found. The ordering of the 4 columns within each of the 48 column-bytes may still be done exhaustively first, but the number of elements in the other ordering problems is too large for that approach.

The Traveling Salesman Problem. The best ordering of the rows or columns is determined by which rows or columns placed next to each other produce the interfaces that maximize the zero groups. Each column-byte edge will produce a particular number of zero pairs when placed next to another column-byte edge. Similarly, each row will produce a particular number of bitline or column address line zero groupings of fours when placed next to another row. If each column-byte is paired with each other column-byte in the data word group, a matrix of the number of non-zero pairs for each match can be produced. Similarly, if each row is paired with all other rows, a matrix of bitline drains remaining and column address drains remaining can be produced. By calculating the distances between all columns in each group and calculating the distances between all rows in the XROM, it is possible to follow the minimum distance path to all column or row nodes and visits each node only once.

The problem of visiting all nodes of a graph only once and traveling the minimum distance is known as the Traveling Salesman Problem (TSP). "The TSP is perhaps one of the most celebrated of all discrete optimization problems" (Parker and Rardin, 1983:69). Since the TSP is an NP-complete problem, many mathematicians, operations researchers, computer scientists and the like have proposed heuristic solutions to it.

The various approaches used to solve the TSP include Dynamic Programming techniques (Held and Karp, 1962), Branch and Bound techniques (Little and others, 1963; Held and Karp, 1970; Dionne and Florian, 1979), Cutting Plane techniques (Gomory, 1958; 1960; 1963; Grotschel and Padberg, 1979), Linear Programming (Dantzig and others, 1959) and many others. So many algorithms have been presented that a number of articles have been written to review them (Bellmore and Nemhauser, 1968; Burkard, 1979; Christofides, 1975; Parker and Rardin, 1983; Held and others, 1984).

The algorithm that has found the proven optimal solution to the largest TSP solved to date (318 cities) was developed by Crowder and Padberg (Crowder and Padberg, 1980) in 1979. Unfortunately, it, like many other TSP algorithms developed, is very complicated and requires an extensive programming effort to implement. However, the first phase of Crowder and Padberg's algorithm implements a straightforward heuristic to obtain a good initial tour for the TSP. This heuristic algorithm was developed by Lin and Kernighan in 1971 (Lin and Kernighan, 1973), but is still an effective and important algorithm for obtaining near-optimal solutions to the TSP (Held and others, 1984). It offers several advantages over the other approaches in solving the TSP. The Lin and Kernighan approach is relatively easy to program and requires no special knowledge of integer or linear programming. The Lin and Kernighan algorithm does not require a lot of memory or CPU time to run large TSP's as does dynamic programming approaches and certain branch and bound techniques.

Since the Lin and Kernighan algorithm for the TSP has been used to obtain near-optimal solutions to problems up to 318 cities and since the algorithm is relatively simple to program, it is the algorithm that was chosen to order the columns and rows of the XROM in order to minimize the number of drains. The Lin and Kernighan TSP algorithm and how it is used to solve the problem of removing drains will be discussed in the next two sections.

<u>The Lin and Kernighan TSP Algorithm</u>. (Lin and Kernighan, 1973). The Lin and Kernighan (L&K) TSP Algorithm is based on the same general approach as their graph partitioning algorithm discussed earlier. It starts with a random TSP tour (a tour is a path that visits all cities/nodes only once, and returns to the starting city/node), and applies an iterative improvement to the

tour by replacing tour links that produce a shorter length tour. When no improved solution can be found, the tour is locally optimum. Like their graph partitioning algorithm, the TSP algorithm attempts to swap a variable number of elements on each pass to obtain a more desirable solution.

The L&K TSP algorithm is described below (see Figure IV-12).

1. Generate a random initial TSP tour.

2. (a) Set i=1

   (b) Select x_i and y_i as the most out of place pair at the ith step. This means the x_i and y_i link are chosen to maximize the improvement when $x_1,...,x_i$ links are exchanged with $y_1,...,y_i$ links. x_i is chosen from the links currently used in the tour, y_i is chosen from the other possible remaining links.

   (c) If it appears that no more gain can be made (according to a stopping rule) go to step 3; otherwise set i=i+1 and go to 2(b).

3. If the best improvement is found for i=k, exchange links $x_1,...,x_k$ with $y_1,...,y_k$ to give a new shorter tour and go to step 2. If no improvement has been found, go to step 4.

4. A local minimum has been found. Repeat from 1 to try and insure a global minimum is obtained.

Much of the power of the L&K Algorithm results from the stopping rule. The algorithm only considers sequences of gains whose partial sum is always positive. This rule can be used because of the following mathematical fact:

If a sequence of numbers has a positive sum, there is a cyclic permutation of these numbers such that every partial sum is positive.

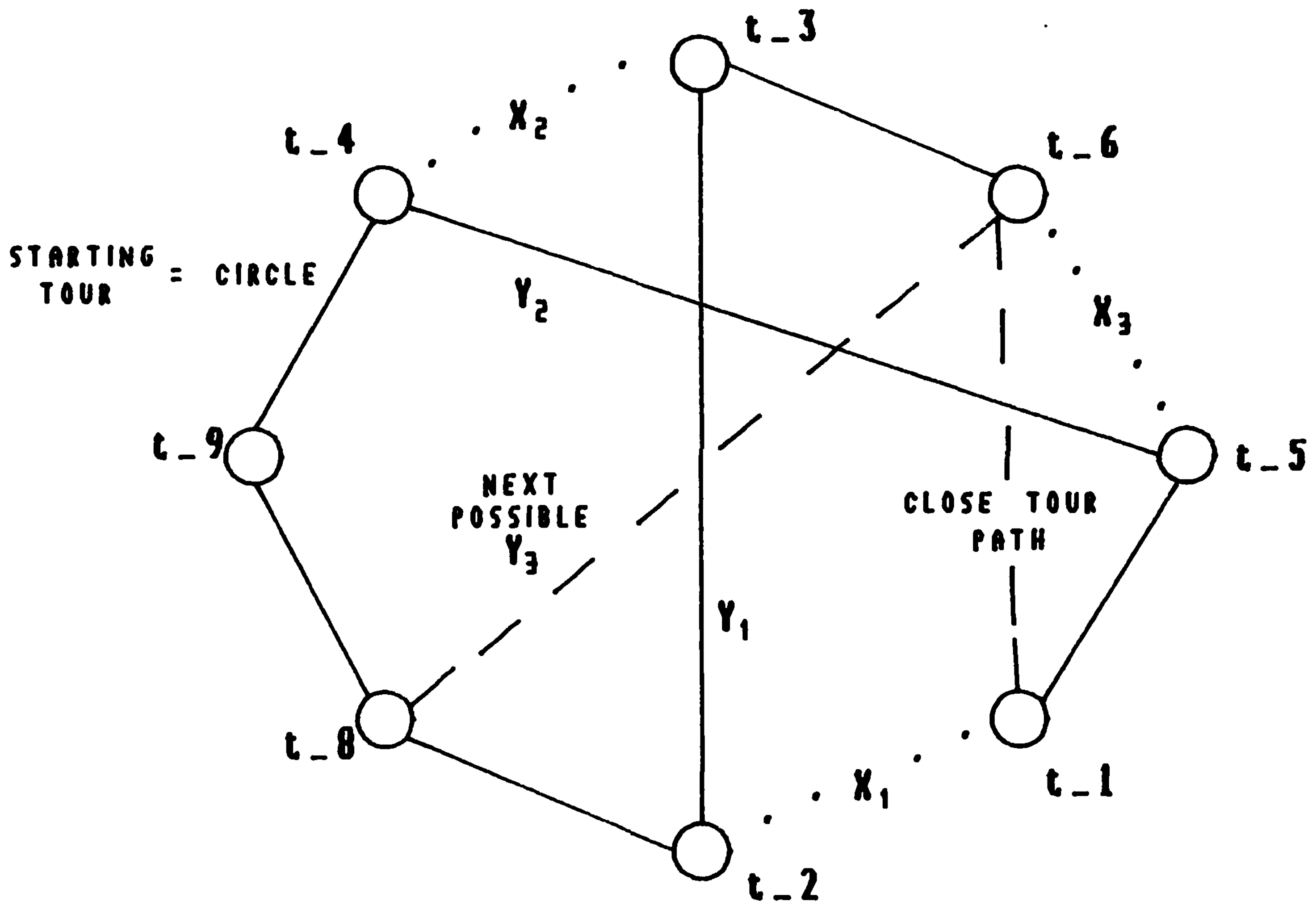


Figure IV-12 Lin and Kernighan TSP Algorithm

This means that choices that exhibit a negative gain at any step in the process need not be considered. The use of this gain criteria greatly reduces the number of sequences to be examined.

Using the L&K TSP Algorithm. The L&K TSP Algorithm is used to minimize the number of drains in the XROM by applying it to the ordering of column-bytes within each of four groups, and then to ordering all of the rows in the XROM. This section will outline the specific procedure used in ordering the column-bytes and rows, and how the L&K TSP algorithm is applied to achieve a near-optimal ordering. The procedure for ordering the column-bytes will be described first, followed by the procedure for ordering the rows.

Column Arranging. Figure IV-13 depicts the layout of XROM cells within a column-byte. There are eight data bits per row, four A0 and four A0N. In reordering these four bitline columns two important facts must be remembered:

1. The address column A0 and A0N lines must stay in place. They cannot be exchanged or two A0 lines (or A0N lines) may become adjacent.
2. When reordering bitlines (which have 2 columns of bit personalization), the new bitline position must provide the same A0 and A0N line orientation as the old position. This insures that bits addressed by the A0 (A0N) line stay addressed by the A0 (A0N) line.

Given the above two rules, each column-byte can take on only four possible column combinations. If the original

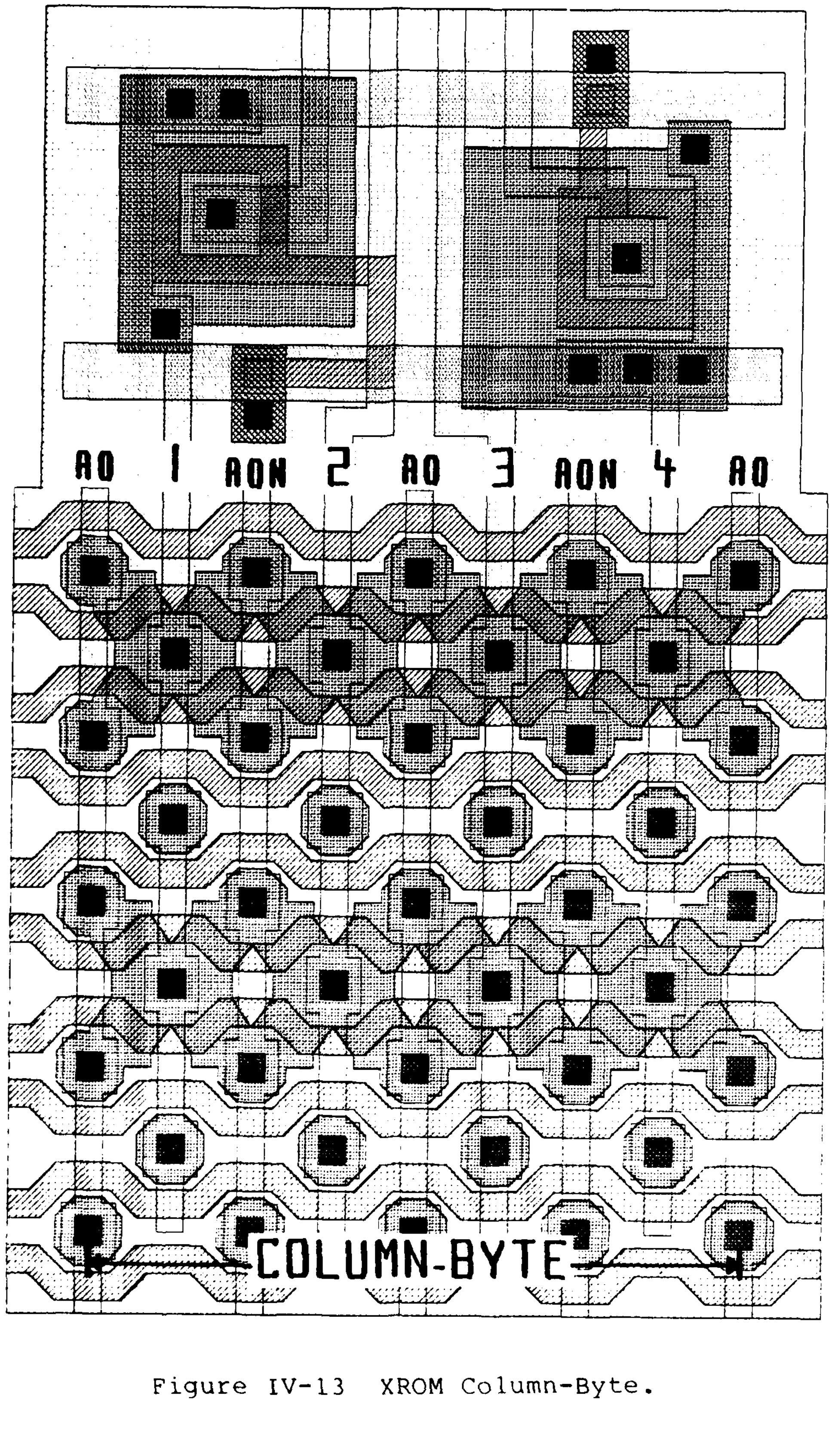


Figure IV-13 XROM Column-Byte.

bitlines are numbered 0 to 3 from left to right, the following Table IV-3 shows the four allowable orderings.

	A0		A0N		A0		A0N		A0
1.		0	-	1	-	2	-	3	
2.		0	-	3	-	2	-	1	
3.		2	-	3	-	0	-	1	
4.		2	-	1	-	0	-	3	

Table IV-3 Four Possible Bit-Column Orderings in a Column-Byte.

As was previously stated, it will not require an extensive amount of CPU time to try all four orderings to obtain the maximum zero pairs between the three interfaces (dashes in Table IV-3) for the four columns. Exhaustively searching the possible combinations for the four columns in each of the 48 XROM column-bytes yields 3 times 48 or 144 maximally zero-paired columns.

The remaining orderings of 12 column-bytes within a data group will be performed by the L&K TSP algorithm based on the zero pairs between column-byte edges. Since the TSP algorithm functions to achieve a minimum "distance" for all 12 column-byte edge links, the appropriate distance measure between edges must be calculated. In this case totaling the number of non-zero pairs produced by the matching of each column-byte edge with all other column-byte edges will provide the distance metric. Notice that the column-bytes are allowed to be mirrored or flipped over in order to make a match. This adds a small degree of complexity in keeping track of the data bits (and final layout), but allows for a better optimization routine. With the

$$2^2 \times((12)^2/2 - 12/2) = 264$$

distance measures calculated for a data group of 12 column-bytes, the L&K TSP algorithm can optimally place the column-bytes to maximize pairs of zeros. Since there are only 12 "cities" in the TSP, an optimal placement is virtually guaranteed for each data word group.

The last step involves changing the TSP tour solution into a Hamiltonian path. A Hamiltonian path is a tour that does not return to the starting city. Since a TSP tour starts and ends at the same node (column-byte), the closed-loop tour must be split. The column-byte edge on the end of each group of 12 column-bytes is not adjacent to the other column-byte edge on the other end. Therefore, if the largest cost link is cut, the optimal zero pair matches remain.

Row Arranging. The procedure to obtain the best ordering of XROM rows is similar to the procedure of ordering the 12 column-bytes described above. The differences result from two factors:

1. A single XROM row of personalization bits does not have two edges. Therefore, the distance calculations need not involve matching two sides of a "node".

2. Unlike the column-bytes, the links of the row TSP must alternate between two different type distance measures.

Figure IV-14 helps show how two different sets of bits of an XROM row are used for two different row matchings: one for zeros about bitline drains and one for zeros about address column drains. Thus, two sets of distance measures between each row and every other row must be calculated. One set for bitline drain removal and one for address column drain removal.

Additionally the L&K TSP algorithm must insure that the links between cities (rows) alternate between bitline drain pairings and address line pairings. Otherwise the L&K algorithm is used as in the column-byte case after the

$$2 \times ((144)^2/2 - 144/2) = 20,588$$

distance measures are calculated.

After the near-optimal ordering of rows to produce the maximum drain removal is achieved by the L&K TSP algorthm, the tour must again be split into a Hamiltonian path. For

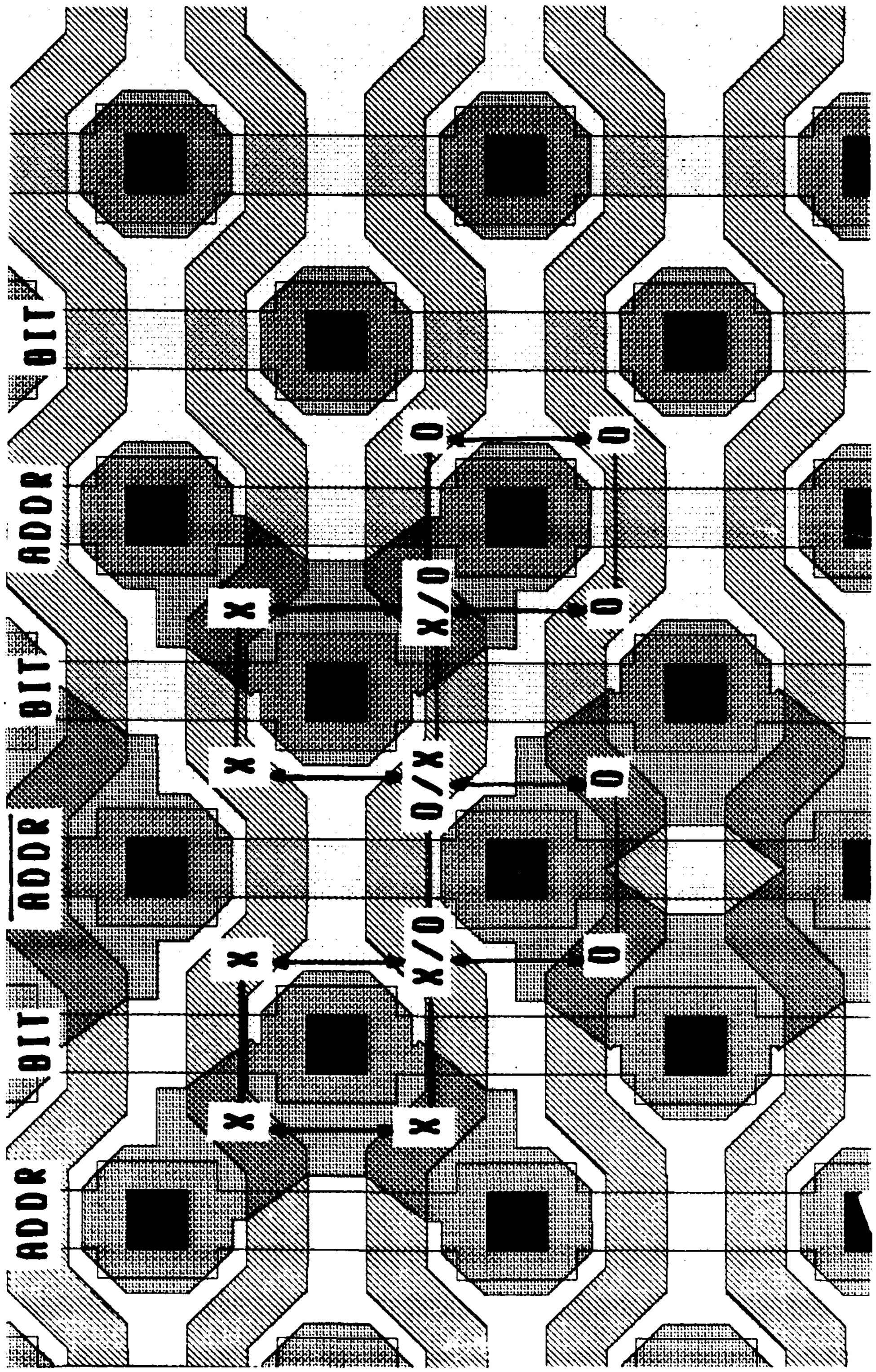


Figure IV-14 XROM Row Matching

the rows, the split will be between the upper and lower rows of the XROM array. In this case, splitting the address column link with the shortest distance will produce more drain removals. This is because only one row borders the extreme upper and lower address column drains of the XROM, and placing the rows with the greatest drain removing potential there will net double the gain.

Since the procedure for minimizing drains in the XROM has been presented, a look at the drain optimization effect on the drain removal for the six data cases of Figure IV-11 is appropriate. Figure IV-15 shows the distribution of drains remaining for the same six cases of Figure IV-11 with the drain minimization procedure applied. The average percentage of drains pulled for the random case is 26.72 percent, and 63.25 percent for the WFTA16. This compares to an average percentage of drains pulled for the random and WFTA16 cases of 6.94 and 24.71 percent respectively for the non-optimized XROM. Table IV-4 summarizes the number of drains remaining for each stage of the optimization procedure on the six XROM test cases. The results of Table IV-4 are shown graphically in Figure IV-16. As with the device minimization, all cases (with the possible exception of the random data case) show significant reductions in the number of drains.

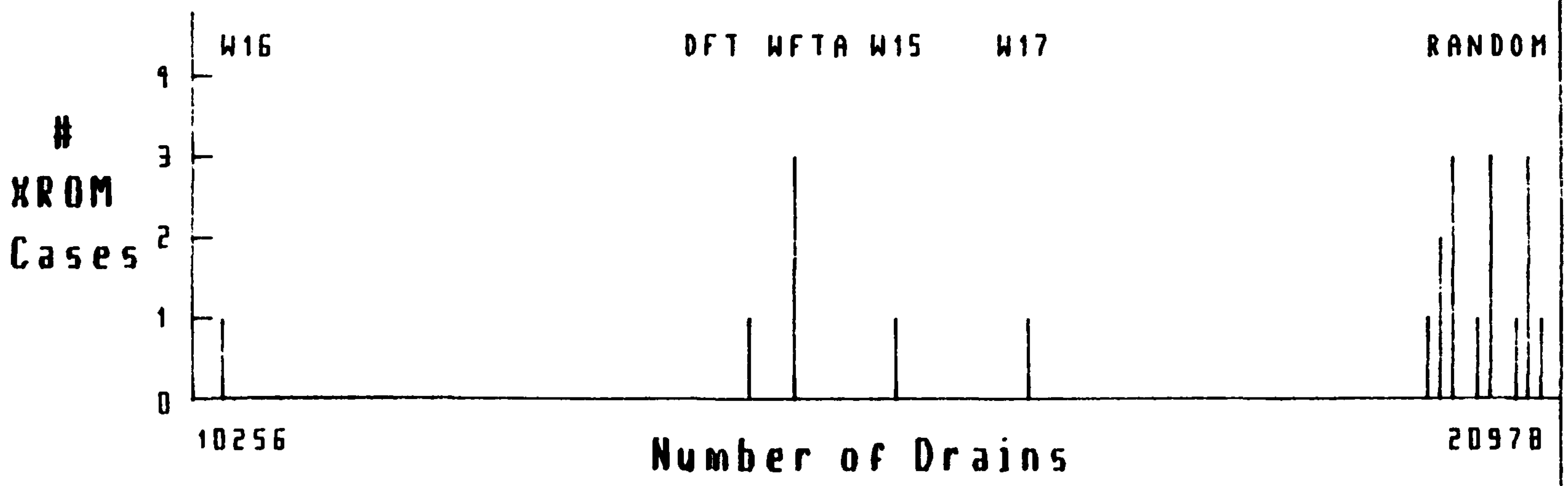


Figure IV-15 Distribution of Drains:
Drain Minimized

STAGE \ CASE	W15	W16	W17	DFT	WFTA	Random
Non-optimized	24054	21017	24248	23733	24645	25974
Device Minimized	19792	15616	20563	18565	18437	24786
Drain Minimized	15616	10256	16732	14502	14870	20455

Drains

Table IV-4 XROM Drain Minimization Summary

XROM Optimization Software

The two programs described in this section implement the device and drain minimization of the XROM. Both programs are coded in the C programming language (Kernighan and Ritche, 1978).

"Placement" - Place and Minimize Devices. The "Placement" program implements the following:

1. Reads in the file containing the desired XROM data words.

2. Performs the following for every possible addressing scheme (105 for 16-point WFTA chip XROM).

    (a) Applies the column sign bits to minimize devices.

    (b) Applies the Kernighan and Lin graph partitioning algorithm to group the columns that are highly correlated for device minimization.

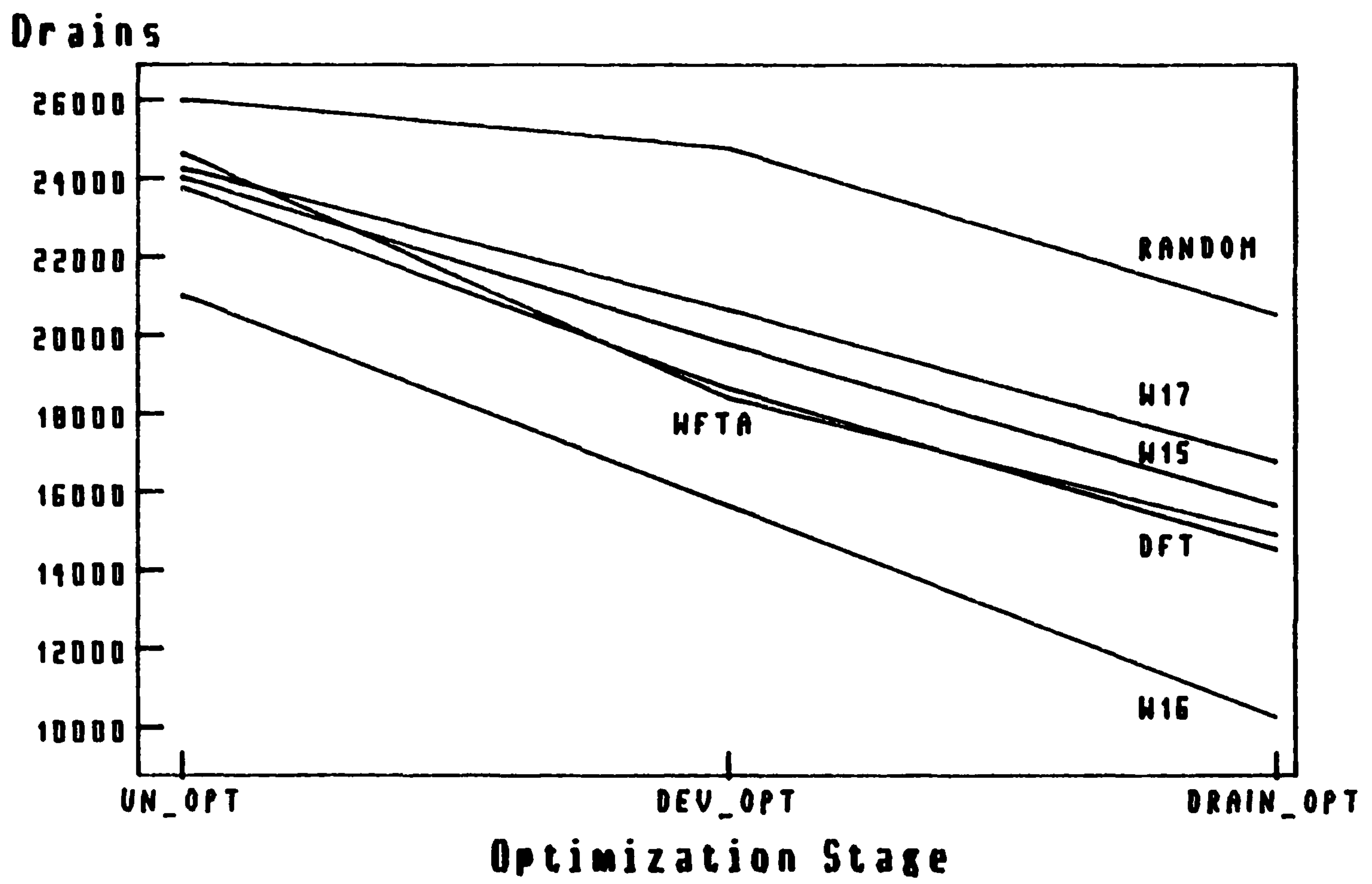


Figure IV-16 Drain Minimization Results

(c) Applies row word sign bit to minimize devices.

(d) Attempts to improve device count by reapplying (b), (c), and (a) in that order.

(e) Saves the XROM addressing scheme, column order, sign bits, and XROM contents if the number of devices is the lowest seen thus far.

3. Restores the best solution so the "Drains" program can continue using the device minimized XROM.

The structure charts for the "Placement" program are depicted in Figures IV-17 to IV-20. The "Placement" program source listing can be found in Appendix B.

"Drains"-Minimize Drains. The "Drains" program implements the following:

1. Calculates the distance measures for all possible column and row pairings.

2. Arranges the columns in each data word group to produce a maximum number of zero pairs via the L&K TSP algorithm.

3. Uses the L&K TSP algorithm to arrange the rows of the XROM to maximize the number of drains pulled.

The structure charts for the "Drains" program are depicted in Figures IV-21 to IV-25. The "Drains" program source listing can be found in Appendix C.

Optimization Results for WFTA16 XROM. This section will present the theoretical power, speed, and yield results for the WFTA16 XROM obtained from the two minimization programs.

Devices and Drains. The number of devices and drains for an optimized WFTA16 XROM are compared to the numbers for the non-optimized version of the same XROM in Table IV-2 and IV-4. As explained previously, these gains

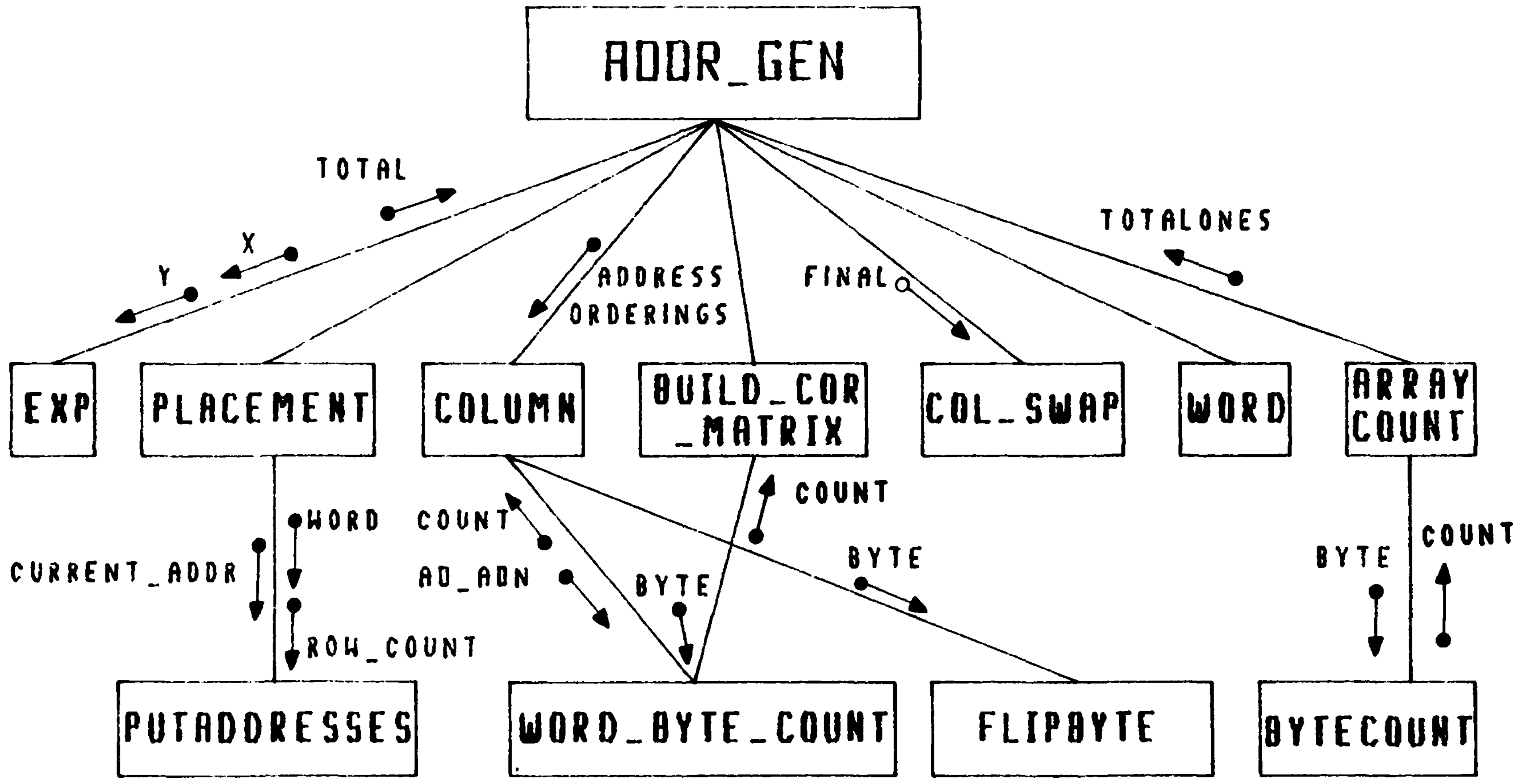


Figure IV-17 'Placement' Structure Chart I

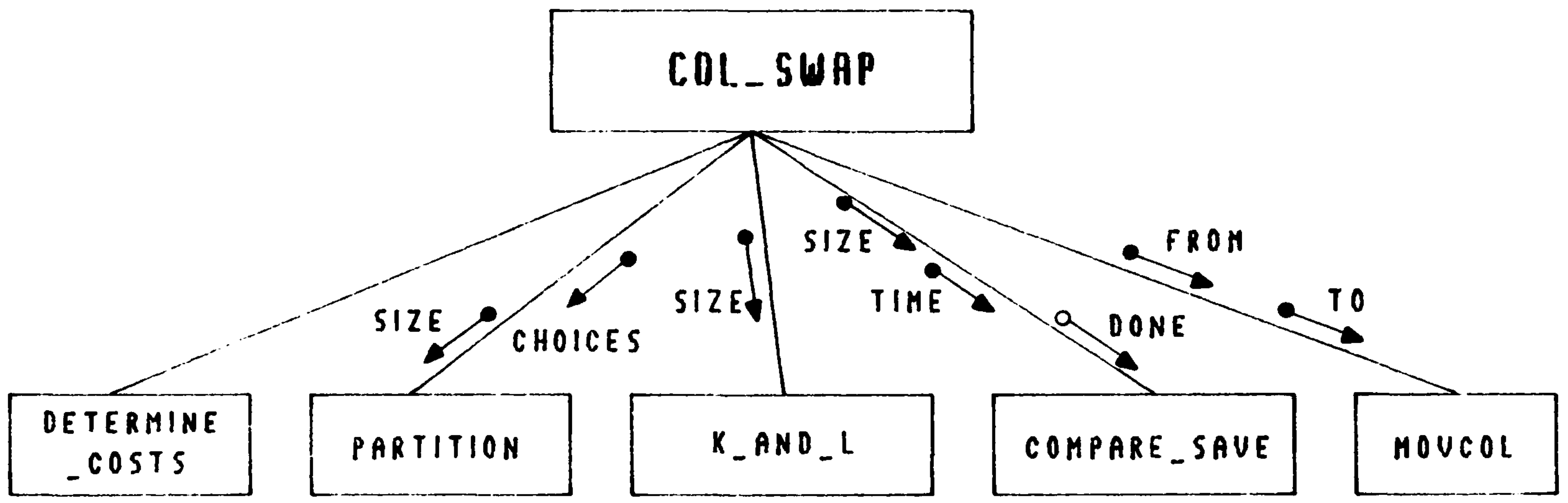


Figure IV-1B 'Placement' Structure Chart 2

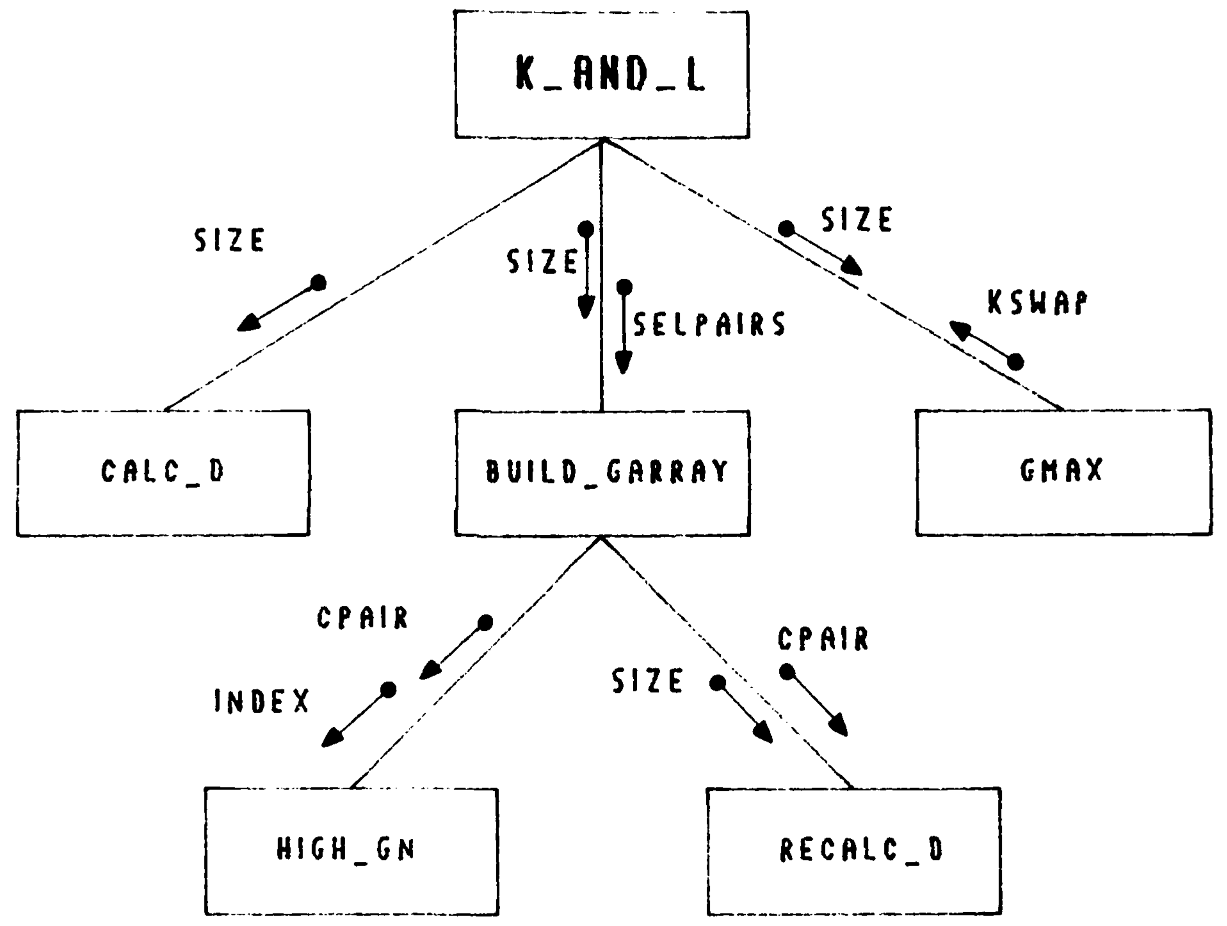


Figure IV-19 'Placement' Stucture Chart 3

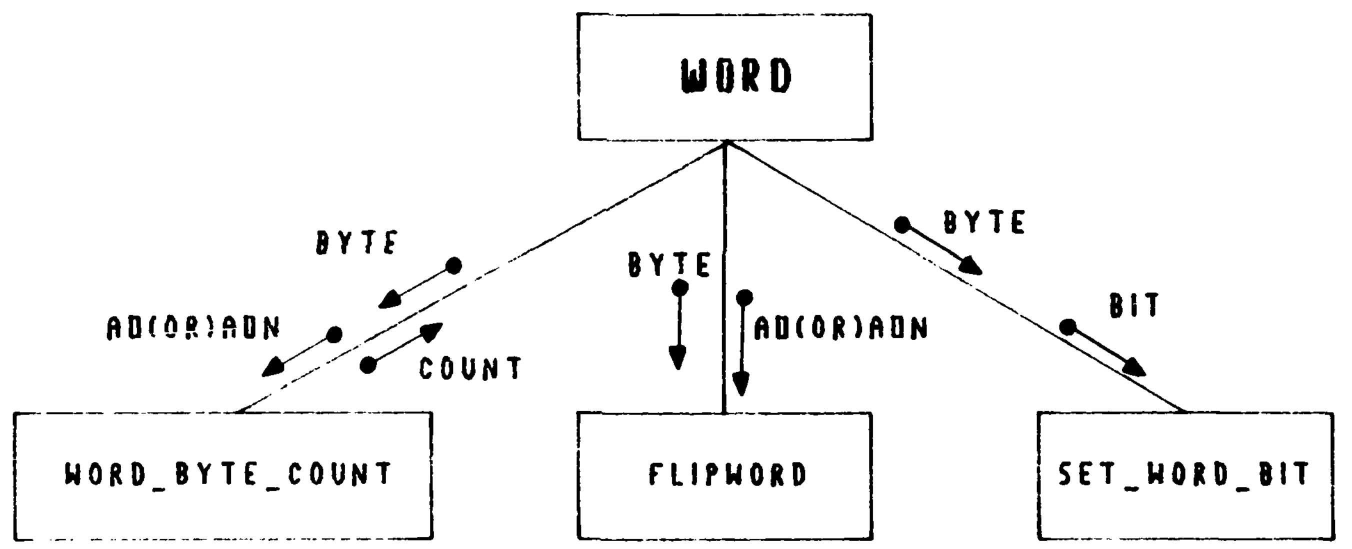


Figure IV-20 'Placement' Structure Chart 4

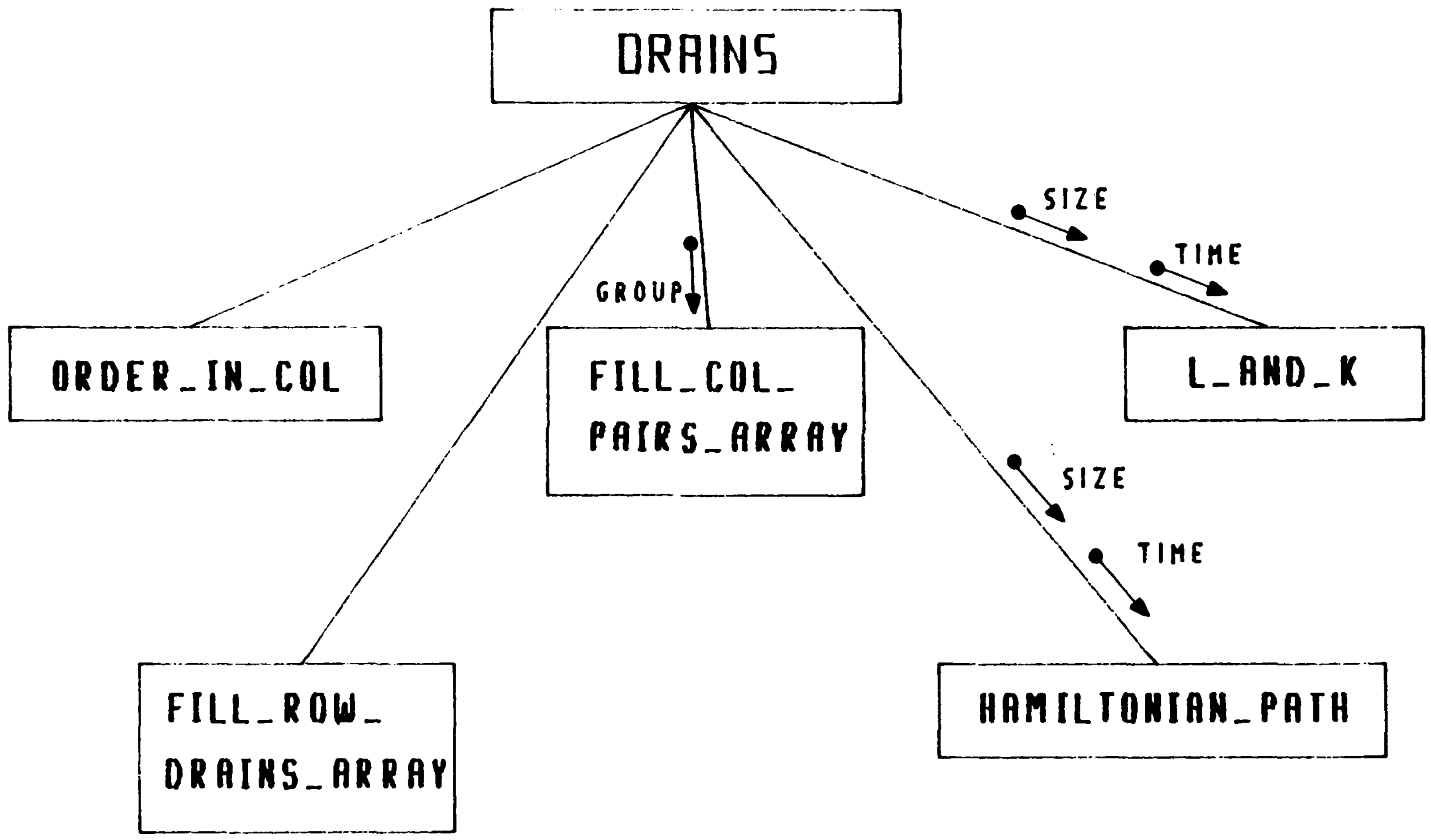


Figure IV-21. 'Drains' Structure Chart I

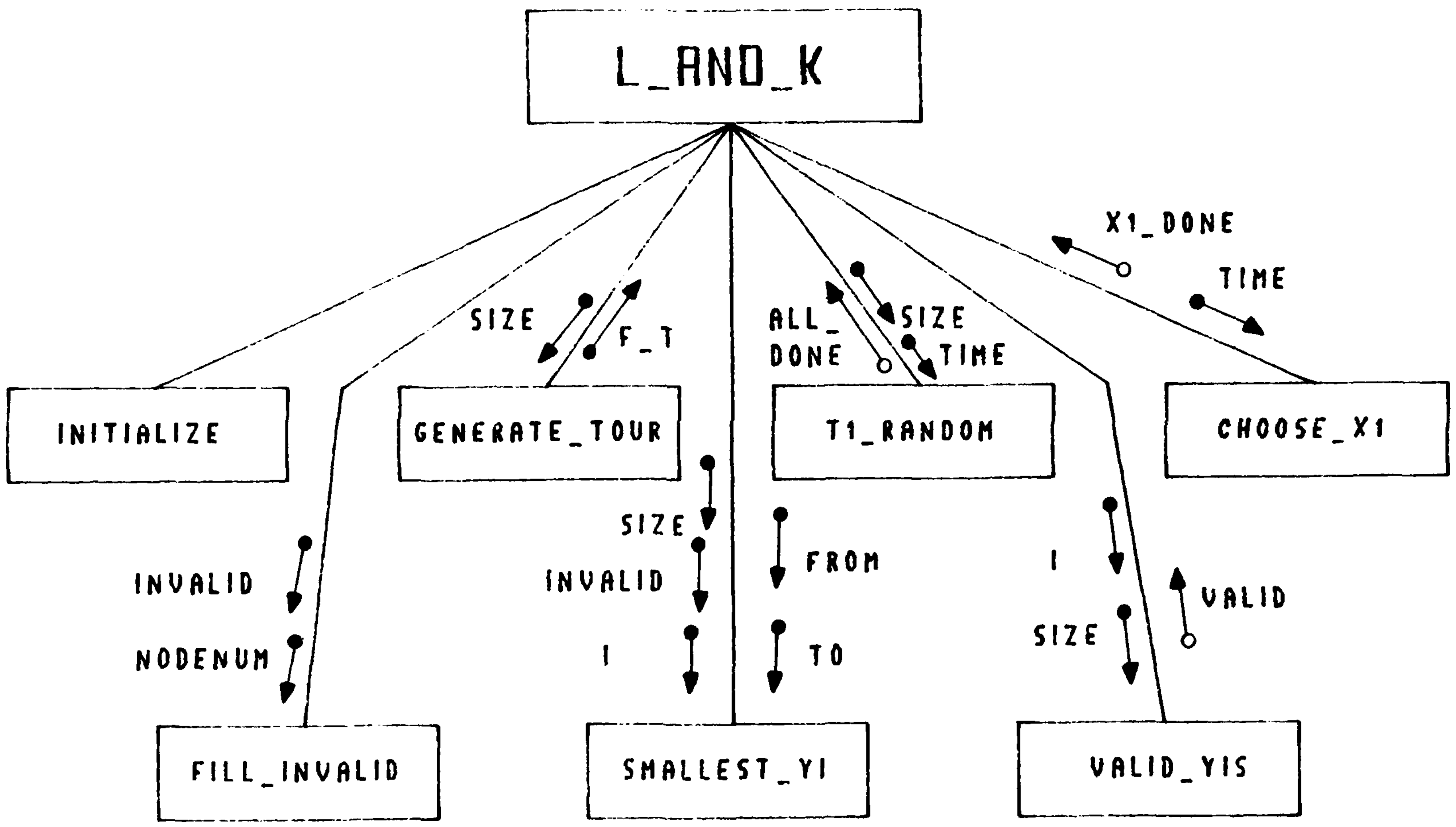


Figure IV-22 'Drains' Structure Chart 2

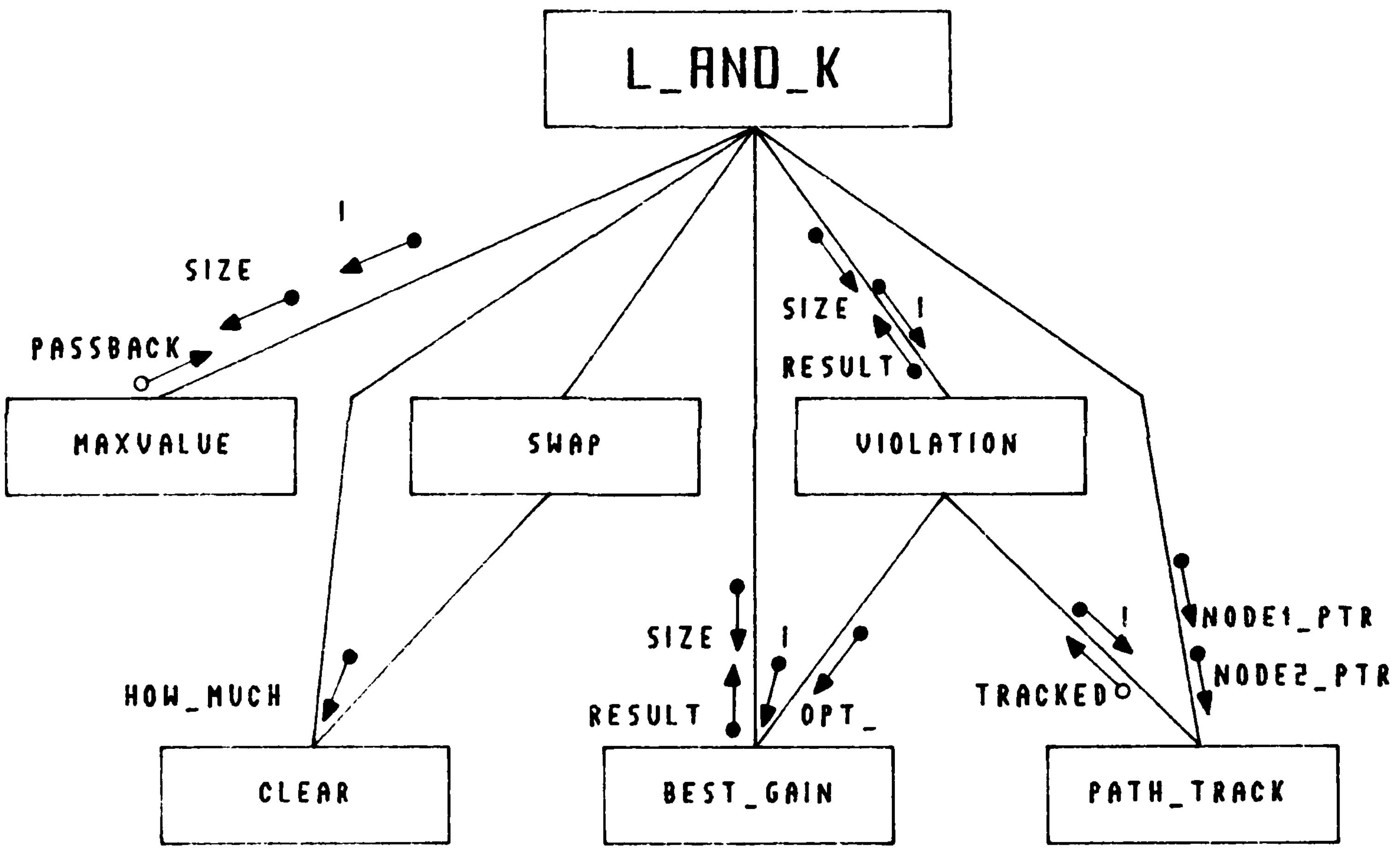


Figure IV-23 'Drains' Structure Chart 3

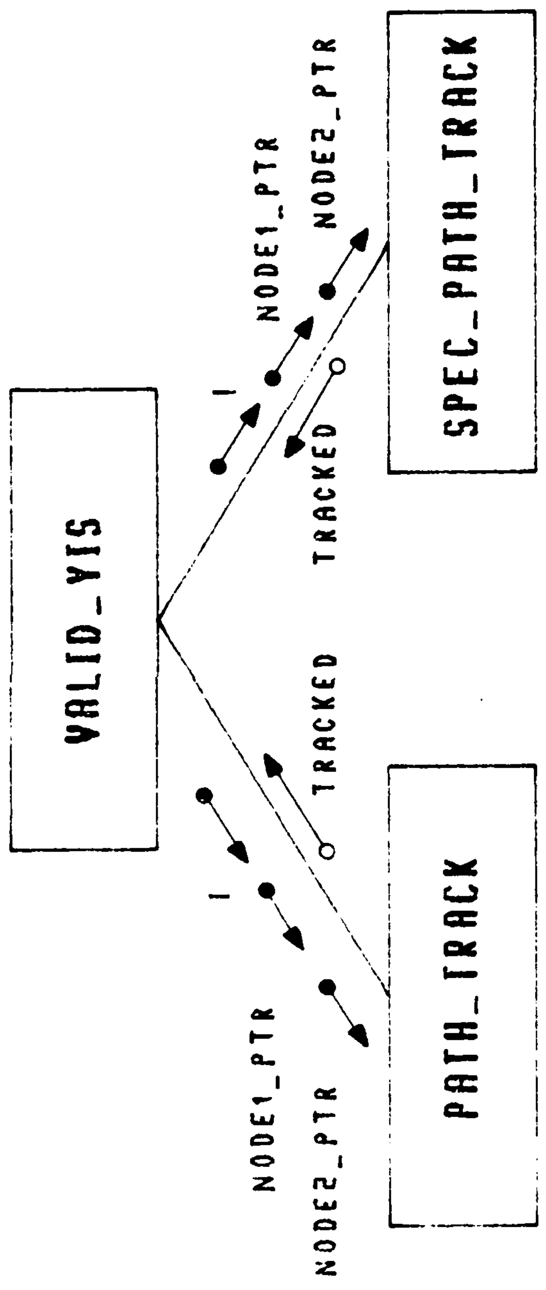


Figure IV-24 'Drains' Structure Chart 4

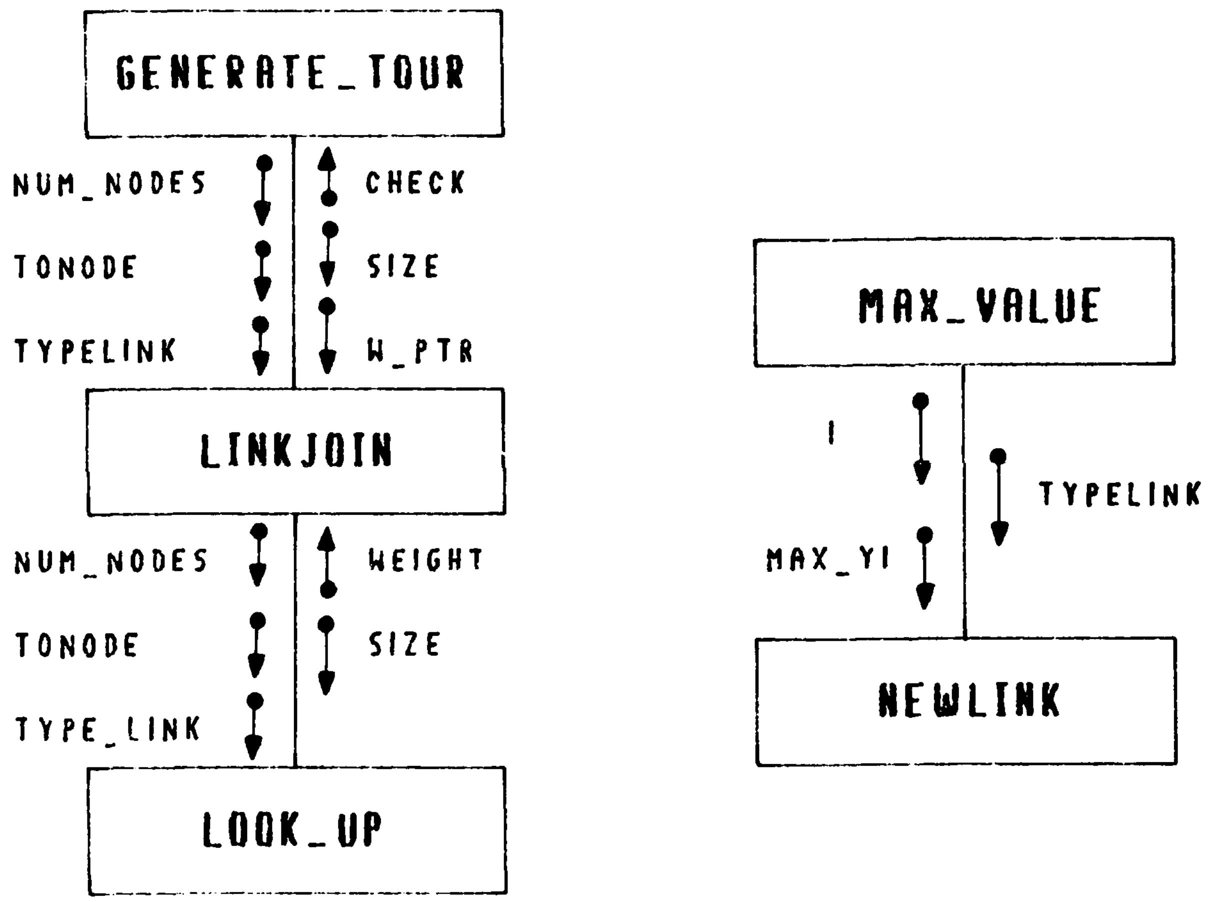


Figure IV-25 'Drains' Structure Chart 5

will improve the power, speed, and yield the XROM. The improvements for the optimized WFTA16 XROM are described in the following sections.

Power. In order to determine the theoretical gain in power dissipation of the optimized XROM over the standard XROM, a "rule of thumb" power equation must be developed. The starting point for this development will be the general equation for switching power dissipation of

$$P = C \times V^2 \times f.$$

where P is the switching power consumed, C is the capacitance of the switched line, V is the voltage level extremes (5 volts for the CMOS XROM), and f is the clocking frequency. If this equation is applied to the average switching path of the XROM, a "rule of thumb" equation can be determined.

The switching path of the XROM (excluding elements that do not change as a function of optimization) can be broken up into five main areas. If an equation for switching power in terms of the number of devices and drains in the XROM can be derived for each area, the sum of these equations will provide the "rule of thumb" total equation desired. Figure IV-26 is a schematic representation of the switching path. The five nodes that comprise the main areas to determine the power used are labeled.

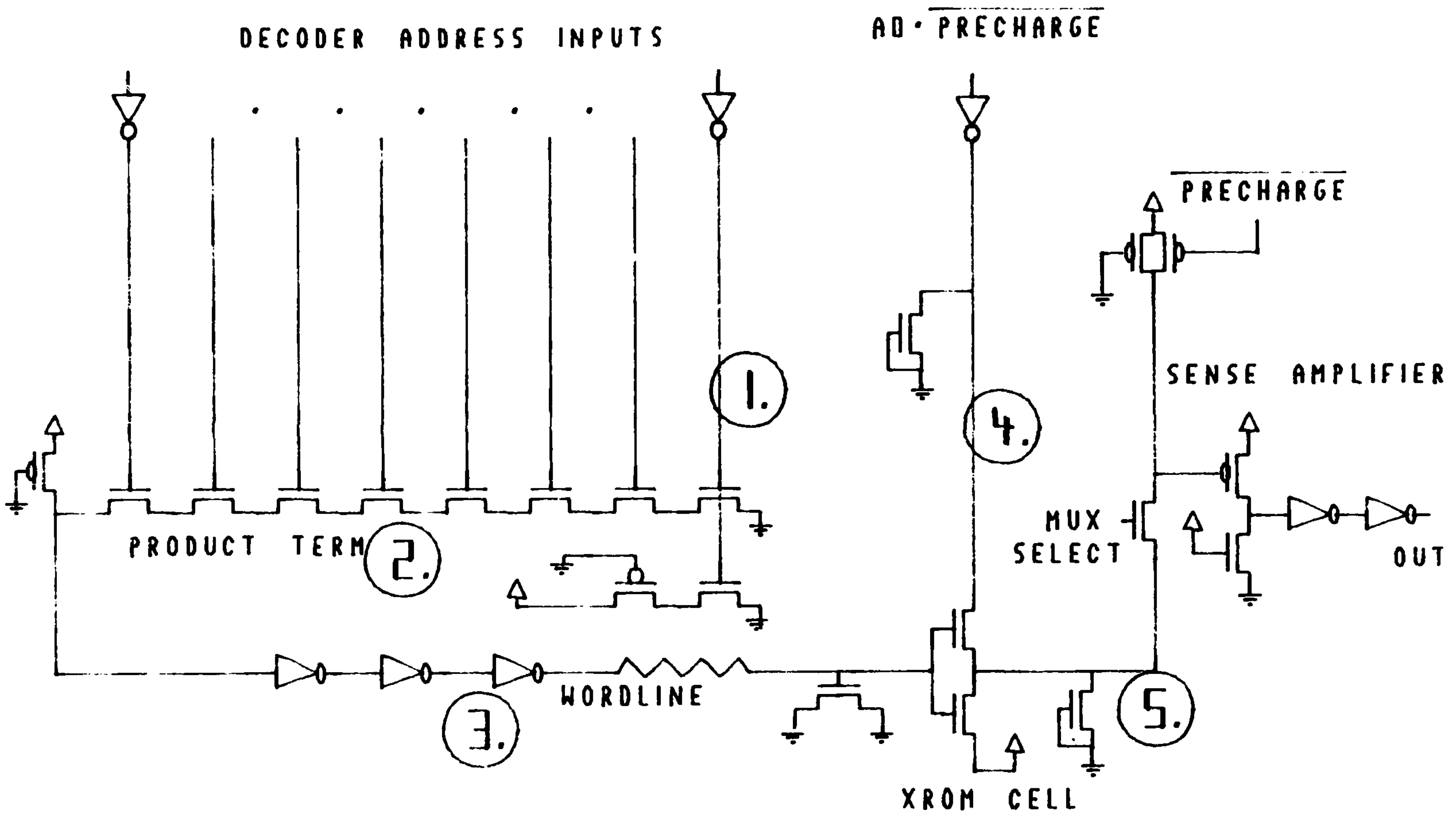
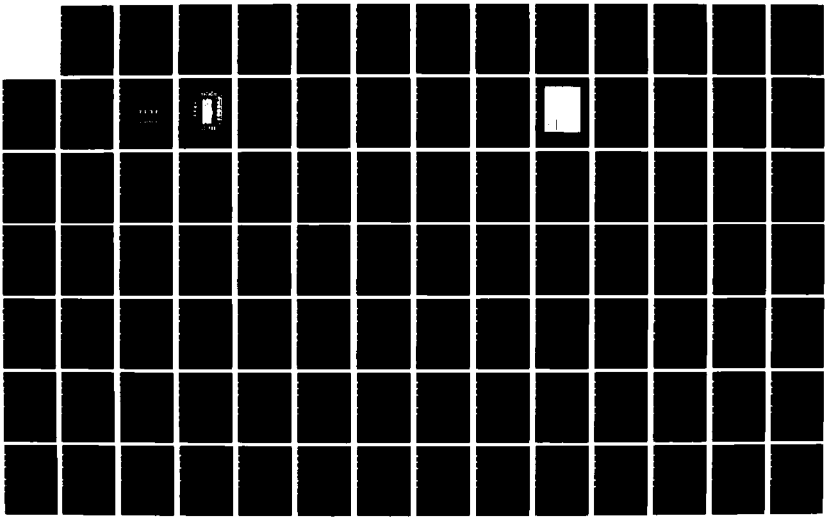
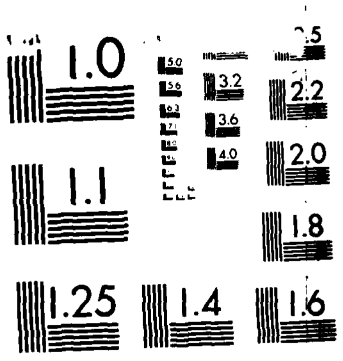


Figure IV-26 XROM Switching Path Schematic

AD-A164 043 CONTROL CIRCUITRY FOR HIGH SPEED VLSI (VERY LARGE SCALE 3/4
INTEGRATION) WINO.. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. P C ROSSBACH
UNCLASSIFIED DEC 85 AFIT/GE/ENG/85D-35 F/G 9/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
3.2
3.6
4.0

The following derivation of power dissipation will refer to Figure IV-26. The derivation will assume that all device sizes used in the XROM are as built in the cells described in this thesis (Chapter III). The result will be but an approximation of the actual power dissipation. The following variable conventions will be used:

P	-	Power (watts)
V	-	Voltage Switching Range (volts)
f	-	Clocking Frequency (Hz)
C_g	-	Gate Capacitance (Farads/micron^2).
C_{ps}	-	Polysilicon to Substrate Capacitance (Farads/micron^2).
N_d	-	N+ diffusion junction area capacitance (Farads/micron^2).
M	-	2nd Metal to diffusion capacitance (Farads/micron^2).
A	-	Average number of PLA decoder addresses.
B	-	Number of bits in the data word.
R	-	Rows in the XROM.
O_T	-	Total ones in the XROM.
D_T	-	Total Drains in the XROM.
S_x	-	Size of the XROM (Bits).

The power used for the first area, the PLA decoder column lines, does not depend on the number of devices or drains in the XROM. The average power consumed is more a function of how many address lines change (on the average) between each data access. The WFTA16 XROM is addressed sequentially, and thus, it is often (50% of the time) the case that only the LSB line of the address is changing. The random datacase may not have a particular addressing sequence, and thus half of the addresses' PLA decoder column lines may be changing. The power consumed in this area is also a function of the number of rows in the XROM.

The number of rows affects the length of the PLA decoder column metal line, and the number of a gates attached to it.

The power equation for the PLA decoder column lines is then,

$$P_d = 4\ A\ C_d\ V^2\ f$$

where

$$C_d = R(108M + 37\ Cg).$$

The constants are determined by the actual decoder layout.

The power used in the second area, the product term line, will be a constant since it depends only on the size of the transistors in the PLA decoder/wordline driver circuit. Additionally only one set of product terms will change on any access. The power equation for the product term line is

$$Pp = 2\ Cp\ V^2\ f$$

where

$$Cp = 54\ Cg + 880\ Nd.$$

The power used in the third area, the wordline, is a function of the average number of devices and drains per XROM row and the number of data bits in the XROM data word. The number of devices (ones) on the row affects the wordline's gate capacitance. The number of drains per row and the number of data bits per data word affects the length of the polysilicon wordline. The drains per row affect the wordline length because a no-drain XROM cell's polysilicon wordline section is shorter than any other XROM cell.

The power equation for the wordline is

$$P_{WL} = 2\ C_{WL}\ V^2\ f$$

where

$$C_{WL} = 10.5(C_g O_T/R) + 9C_{ps}[6(2B + 1) + D_T/R].$$

The power used in the fourth area, the address column lines (A0 and A0N), is a function of the number of drains on the line, the number of lines (determined by the number of bits in the data word, B), and the length of each line. The column address lines are precharged high on each clock cycle and one half are pulled low on each access. Thus, on the average 1/2 the lines will change state each cycle. The length of each line is determined by the number of rows in the XROM, and the average number of drains on each line is determined by the total number of drains.

The power equation for the address column lines is

$$P_{ac} = 8BC_{ac}V^2f$$

where

$$C_{ac} = 27RM + 9D_T\ N_d/8B.$$

The power used in the last area, the bitlines, is a function of the same capacitance factors as the address column lines. The differences lie in the number of bitlines that will change state each access and in the voltage swing of the bitlines. The number of bitlines that go high on an access may be less than half for the optimized XROM. For a non-optimized XROM with random or sequential data entries, the number of bitlines that change state each cycle is expected to be half the total. After optimization it is expected that more bitlines will remain unchanged on each cycle. This number of bitlines that remains unchanged is a function of the number of devices in the XROM. Since the bitlines are precharged through an n-type device, they only reach 3.3 volts. Also, if the XROM is operating at full speed, the bitlines may not be pulled down all the way to zero. Thus, a voltage swing of 1/2V will be used.

The power equation for the bitlines is

$$P_{BL} = 4BO_TC_{ac}V^2f/S_X.$$

Therefore, by summing all five area power equations the "rule of thumb" XROM power equation becomes,

$$P_T = 2V^2f[2AC_d + C_p + C_{WL} + 4BC_{ac}(1 + O_T/2S_X)].$$

A good approximation of the power savings produced by XROM optimization can now be calculated. Using the parameters listed in Appendix F for the 3 micron CMOS process, the results can be calculated for the gain realized in the WFTA16 XROM. Using the 54K XROM size, the number of devices and drains as given in Table IV-2 and IV-4, and a clocking frequency of 12.5 MHz, the power gains are calculated. The calculated power consumption is 45mW for the non-optimized version of the 54K XROM, and 24mW for the optimized XROM. These results show a power savings of approximately 50 percent.

Speed. The improvement that is realized in the area of XROM speed of operation can be simulated using SPICE. This approach will produce more accurate results than attempting to present a general equation for speed. The key items in the SPICE simulation that could improve the speed for the optimized XROM are the decrease in polysilicon wordline length, the decrease in gate capacitances on the wordline, and the decrease in drain capacitances on the bitline. The wordline length decreases by 2 microns with each no-drain XROM cell that it passes through. The gate capacitances on the wordline decrease with each device removed from its row. The bitline drain capacitances decrease with every no-drain XROM cell on the bitline column. Thus, with fewer devices and drains in the XROM, the worst-case operation of the XROM is expected to improve.

Using the device and drain minimization programs the following speed results can be obtained for the WFTA16 XROM using the SPICE simulator. Table IV-5 shows the gain in switching speed for the XROM's wordlines and bitlines. The sum of these improvements gives the average access time gain of the optimized WFTA16 XROM over the nonoptimized WFTA16 XROM. Speed improvements for any XROM will ultimately depend on how the worst-case path through the XROM is affected by the optimization routine. The WFTA16 XROM did not achieve a significant gain in speed because the worst-case wordline and bitline capacitances were not substantially improved by the optimization procedure.

<u>Yield</u>. The chip yield improvement that the optimized XROM will provide is difficult, if not impossible, to quantify. A number of factors such as final chip size

	Wordline	Bitline	Access
Non-optimized	34nS	22nS	56nS
Optimized	33nS	21nS	54nS
Total Gain	1nS	1nS	2nS

Table IV-5 XROM Speed Improvement.

and area, total number of devices, fabrication process, and many others will have a great impact on the WFTA chip's yield. It is, however, easily seen that the significant decrease in the number of active devices, drain implants, and connections will result in an improvement in the total chip yield. It is expected that the yield improvement for certain type defects will be at least proportional to the reduction in transistors and active drain area. For the WFTA16 XROM, there was a 40 percent decrease in transistors, and a 60 percent decrease in active drain area. These reductions should have a positive effect on total chip yield.

Automatic Layout of the XROM

Generating XROM Caesar Files. The need to automatically generate the personalization cell layout for the XROM was explained in the first section of this chapter. After the XROM personalizations have been optimized for the desired parameters by the two minimization programs, it only remains to output the optimized configuration to some layout description.

Since the WFTA chips were being designed with the Caesar interactive design software, Caesar files were written to describe the XROM personalizations.

The approach taken in generating these output Caesar files was to use cell calls for each personalized XROM cell as stored in the "Drains" results. There are four major XROM areas that require personalization as determined by the "Placement" and "Drains" programs. These are:

1. The sense amplifier array. The sense amplifier's cell configuration is determined by both minimization programs. The "Placement" program determines if the sense amplifier for a given column is to be inverting or not. They are the column sign bits. The "Drains" program may or may not mirror or flip over a column-byte to solve the TSP. If a column-byte is mirrored, the sense amplifier must compensate by rerouting the column bit lines. Thus, four different functional type sense amplifier cells must exist to be called.

2. The PLA wordline decoder. After the "Drains" program solves the TSP for all rows of the XROM, the correct addresses must be personalized on the corresponding wordlines in the PLA decoder. Due to the unique polysilicon personalization (described in Chapter III), two wordline bits are personalized by one Caesar cell. Thus, four different types of PLA wordline decoder personalizations are required for calls in the Caesar file.

3. The four sections of the XROM array itself. The final positioning of the ones and zeros in the XROM obtained from the "Placement" and "Drains" programs determine the final XROM layout. Each XROM cell containing one drain position and four bit positions are placed on a bitline or address column line for every four bits in the array output from the optimizing software. Thus, 16 unique XROM cells are needed for calls in the Caesar file.

4. The word sign bit columns. The word sign bit results from "Placement" and "Drains" must be placed in the correct columns for the XROM. The same 16 XROM cell configurations are needed for the word sign bit columns as for the XROM array.

The XROM layout software determines the particular cell calls to be made for the particular bit pattern produced by the XROM optimization software. A Caesar file for each major section described above is created. Each Caesar file calls the proper personalization cells and transforms the cell to the correct location in the section's layout.

The interfacing between the three programs, "Placement," "Drains," and "Layout", is performed by a small program called "gen_XROM". It simply declares the global variables and calls all three programs in the proper order.

<u>"Layout" - Automatically Generate Caesar Files of the XROM</u>. The "Layout" program, like the other software developed in this thesis effort, was written in the C programming language. The "Layout" program generates the following Caesar files:

1. 4 Sense amplifier arrays. Each array contains the same number of sense amplifiers as bits in the data word. One array is placed over each of the corresponding XROM array groups.

2. 2 PLA wordline decoder personalization arrays. One for the left side of both XROM subarrays and one for the right.

3. 4 XROM array groups.

4. 2 Word sign bit columns. One for the center of each XROM subarray.

The Caesar files described above can be easily used to build the XROM that will generate the original data that was read in by the "Placement" program.

The automatic generation of the Caesar files was successfully demonstrated for a number of test cases, and for the WFTA15, WFTA16, and WFTA17 XROM.

The structure chart for the "Layout" program is depicted in Figure IV-27. The Layout" program source listing can be found in Appendix D.

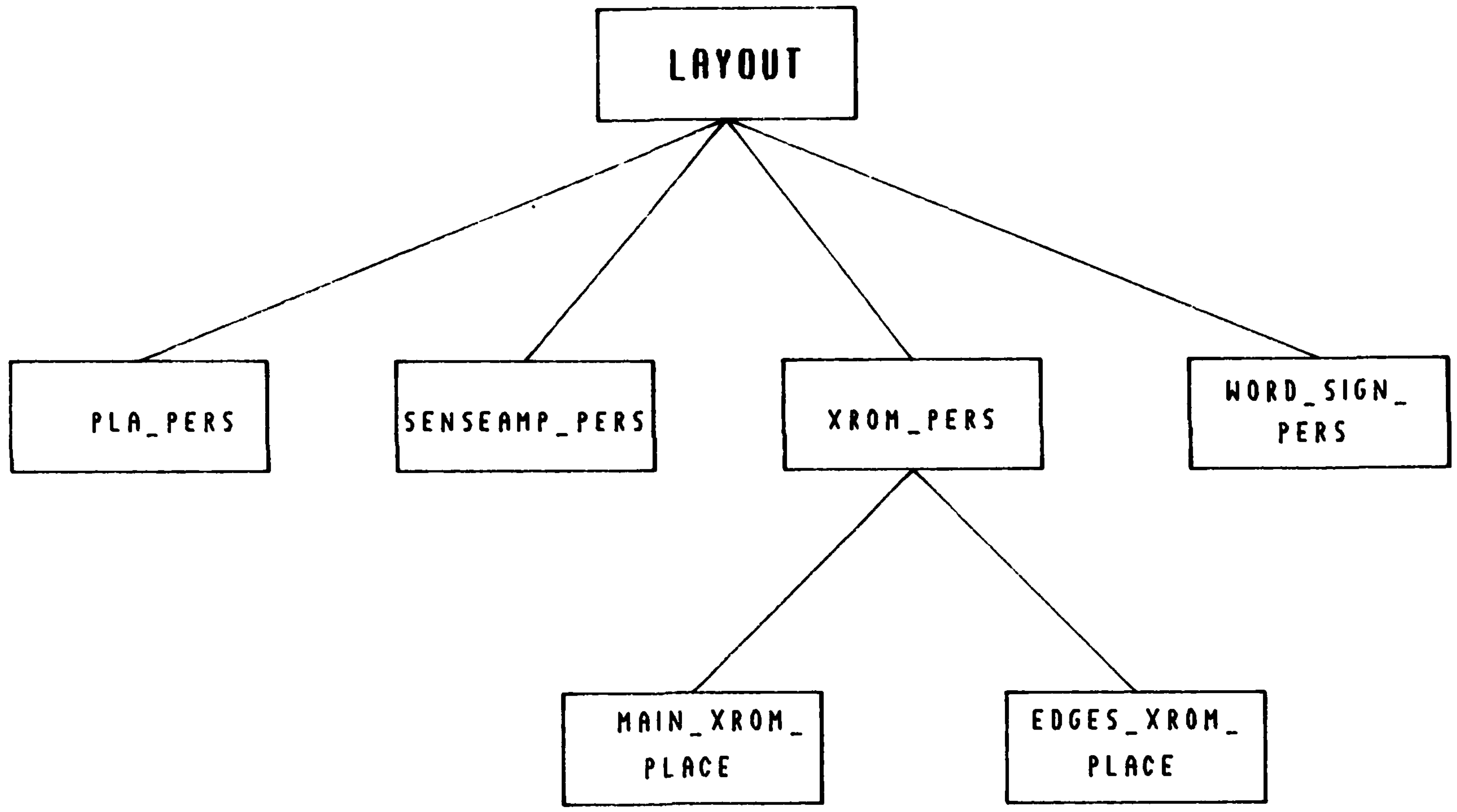


Figure IV-27 'Layout' Structure Chart

V. Fabrication, Testing, and Evaluation

Fabrication

CMOS Fabrication. CMOS circuits can be fabricated using a number of different approaches. Among the most well known are the p-well process, n-well process, twin tub process, and silicon on sapphire or insulator. A commonly used approach is the p-well process. The p-well process starts with a moderately doped n-type substrate in which the p-channel transistors are built, and creates p-type wells in which the n-channel transistors are made. The layout and p-type well process cross-section of a CMOS inverter are shown in Figure V-1. This p-well process is the type of CMOS technology used in the fabrication of the integrated circuits presented in this thesis. The circuits were fabricated through the MOS Implementation Service (MOSIS).

MOSIS Facility. MOSIS is operated by the Information Sciences Institute of the University of Southern California under the sponsorship of the US Defense Advanced Research Projects Agency. MOSIS is a facility that serves as an interface between designers in the academic and industrial communities and the venders that fabricate the devices. Designers submit Caltech Intermediate Form (CIF) layout descriptions of their chips via electronic mail to the MOSIS facility. MOSIS compiles a multiproject wafer and contracts with the semiconductor industry for mask making, wafer

Sequencer test chip was submitted May, 1985 and was received in July, 1985. The XROM Address Generator test chip was submitted in September, 1985. The photomicrograph of the Control Sequencer test chip is included in Figure V-2.

The WFTA16 chip should be submitted in 1986.

Testing of the Control Circuitry

Objective. The objective of the control circuit testing was to determine the functionality, speed, and power dissipation of the Control Sequencer and XROM Address Generator. These results could then be used to validate, or modify as necessary, the control circuits before they were integrated into the 1.2 micron implementation of the WFTA16.

Problems. Since the XROM Address Generator test chip was not received in time to be tested, test results were obtained only for the Control Sequencer test chip. Therefore, the following sections will only discuss the testing and results of the Control Sequencer.

Equipment/Setup. The test setup for the Control Sequencer chip is shown in Figure V-3. The Control Sequencer chip was placed on a breadboard and wired according to the pinout diagram. Power was applied to the chip using a standard 5-volt DC supply. Many of the chip's input pins were controlled using simple switches because fabrication, and packaging. MOSIS then delivers the packaged integrated circuits (ICS) to the designer.

Although this method of chip fabrication was satisfactory, two major problems were experienced. The first was an inability to successfully transmit CIF files to MOSIS from AFIT. Some fabrication submission dates were missed when test chip CIF files failed to arrive at MOSIS. It is still not known why these failures occurred. The second problem is that MOSIS contracts numerous companies to fabricate the ICs and each has its own process parameters. Therefore, the speed of the circuit as tested may be significantly different from the speed that could be achieved with a different fabricator.

Process and Parameters. The test circuit chips of the Control Sequencer and XROM Address Generator were designed and fabricated using a scalable, 3 micron, CMOS process. The scalable process facilitates processing the same circuit layouts (except pads) at both 3 microns and 1.2 microns. Thus, no major cell changes are needed in transitioning to the 1.2 micron implementation of the WFTA16.

A typical set of SPICE parameters for the 3 micron CMOS process is given in Appendix E. Ranges of parasitic capacitances for the 3 micron CMOS devices are given in Appendix F.

Submissions. Two control test chips were submitted for fabrication during the WFTA16 design phase. The Control

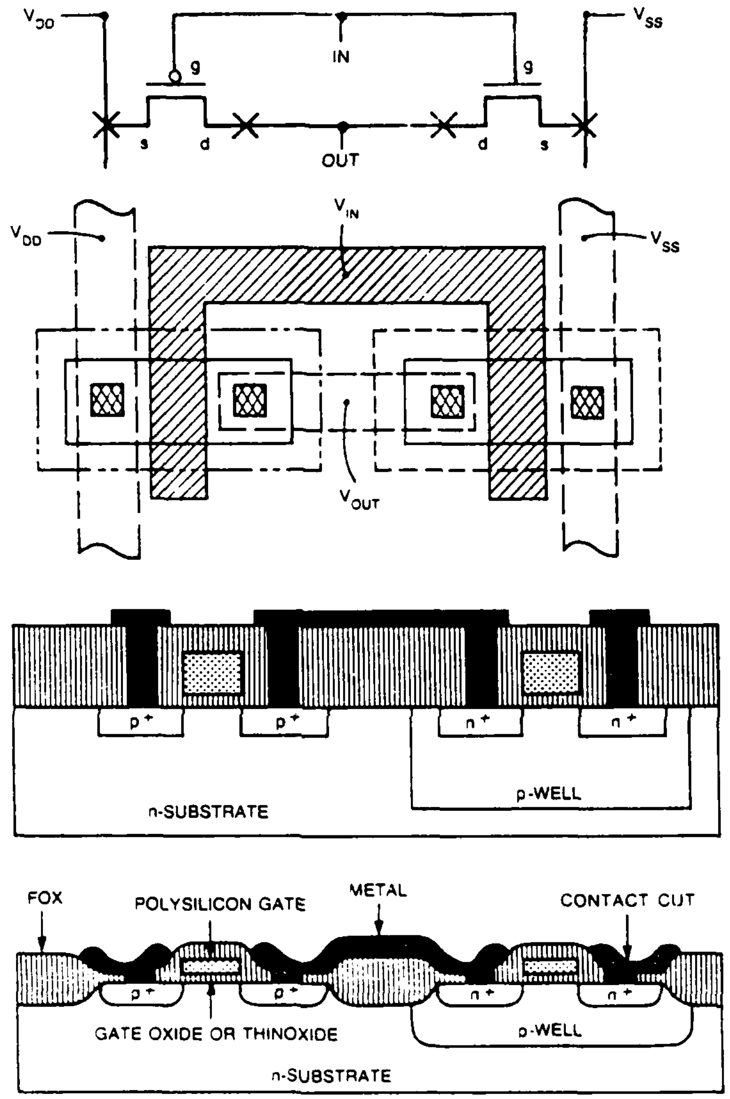


Figure V-1 P-Well CMOS Process.

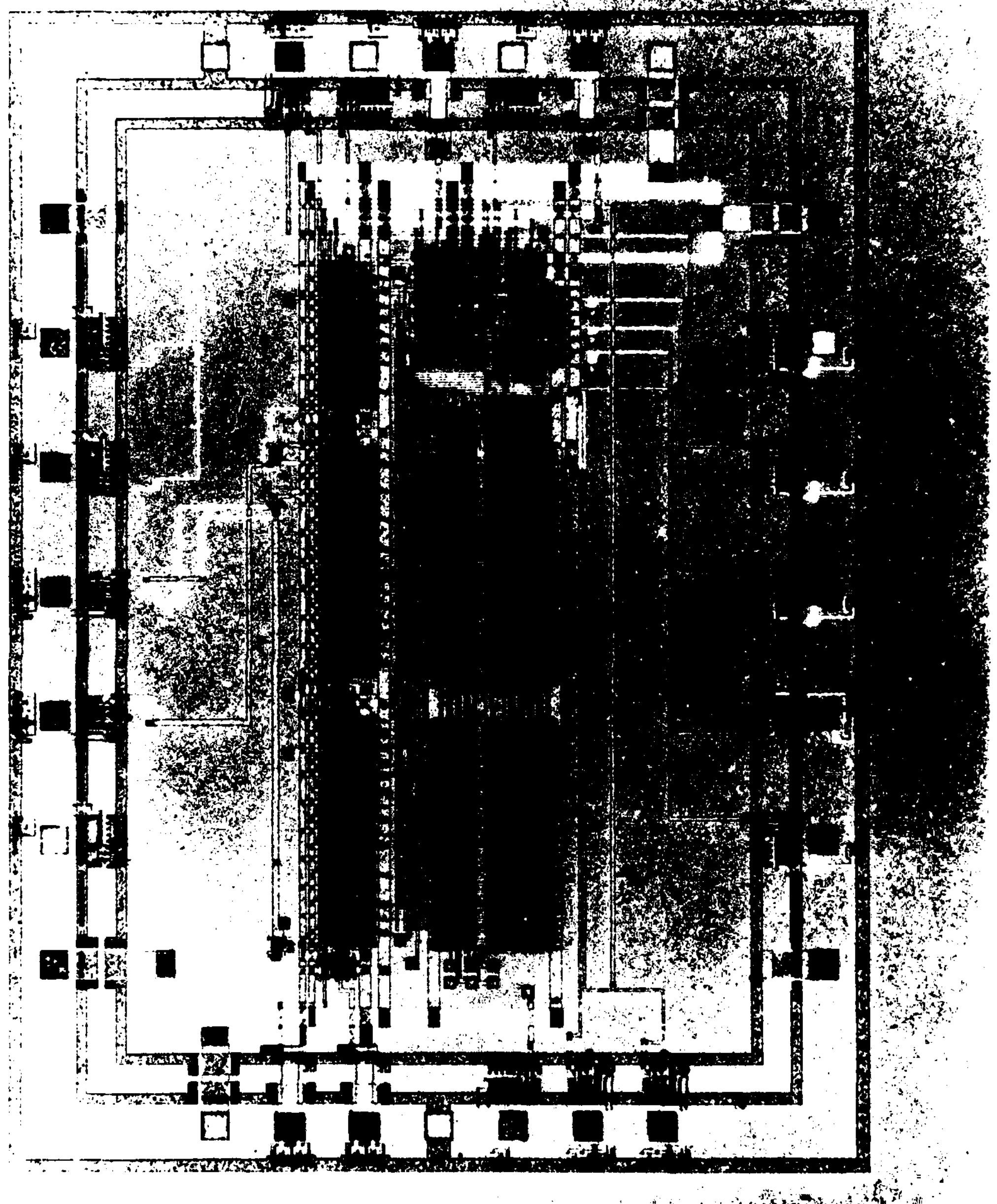


Figure V-2 Control Sequencer: Photomicrograph.

they operated asynchronously or because they stayed at one level during a test.

When performing static, functional tests on the circuit, push button switches were utilized to generate the two clock phases, phi 1 and phi 2. When performing dynamic testing for speed, a programmable, eight-output, 50 MHz, signal generator was used to generate a two phase, non-overlapping clock signal.

Output signals were observed using 1 GHz and 80 MHz oscilloscopes. The test setup was implemented to observe the outputs as they were driven off-chip. If the output drivers limited the chip's speed of operation, the programmable signal generator would be used to clock the circuit the exact number of times to bring an output signal to a pad and stop. At this point, the outputs could be read to see if the circuit functioned to speed. This approach can be used since the Control Sequencer's output signals are not driven off-chip in the WFTA processors.

Procedure. A functional block diagram of the Control Sequencer test chip is shown in Figure V-4. Due to the relative simplicity of the Control Sequencer's operation (described in Chapters II and III), a straightforward test procedure was used. With power and clock signals applied to the chip, the operate line was raised, and the nine output pulse trains were observed for proper functionality at the output pins. As long as the continue input was high,the

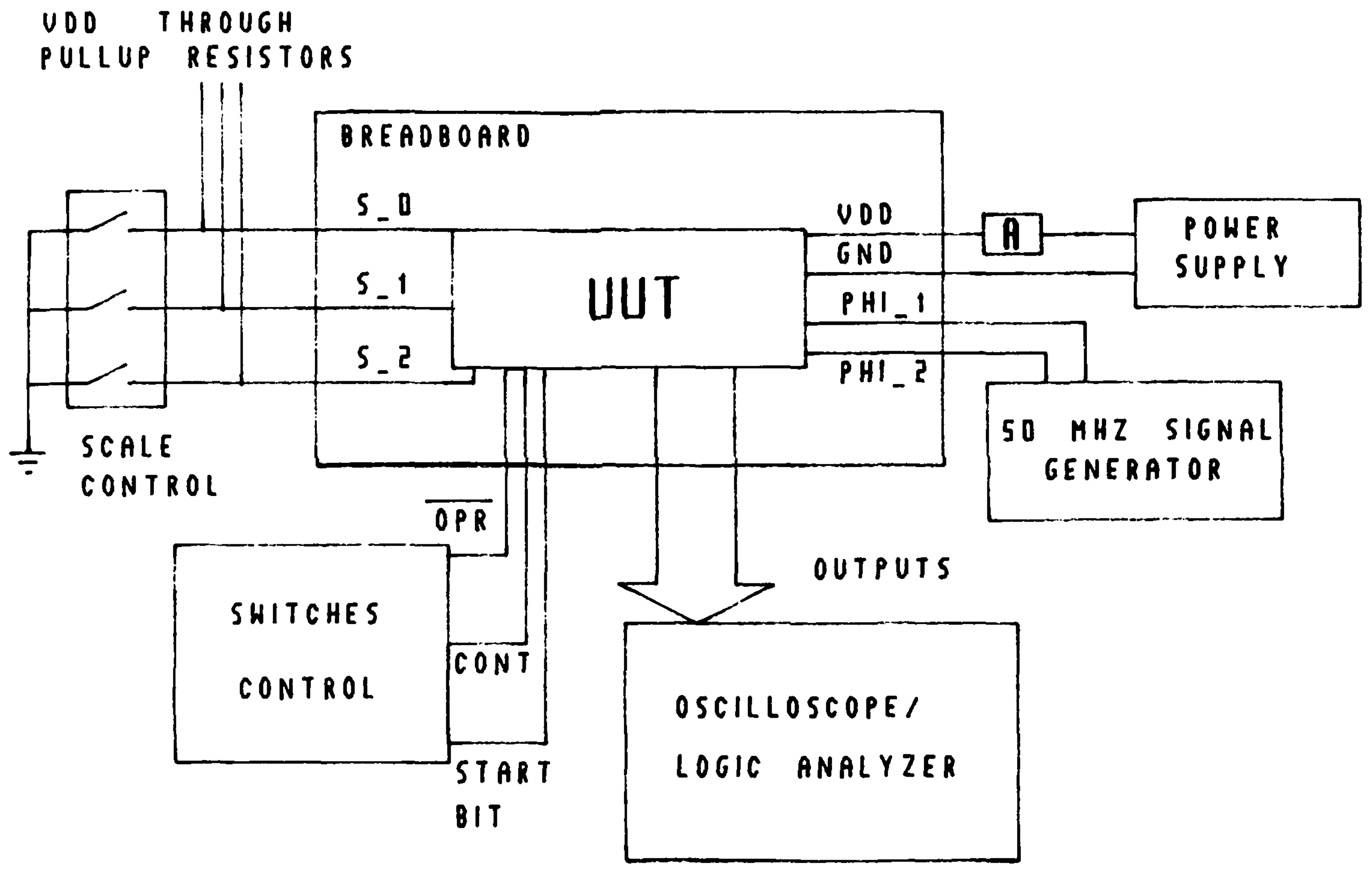


Figure V-3 Control Sequencer Test Setup

outputs were to be periodic every 32 clock cycles. The three-bit, scale factor input was changed and the pulse widths of the output signals (especially those that are a function of scaling) were observed for proper functionality.

This procedure was followed for both low speed/static and high speed/dynamic testing. If proper functionality could not be achieved for the chip, numerous controllability and observability signals could be exercised to help ascertain the problem with the control circuit. Test vectors could be loaded into the ring counter using the start_bit pin, and the PLA results read out from either the output pads or numerous probe pads placed on the chip. The bit_bottom pin and probe pads could be used to see if the bit was propagating down the ring counter. These and a number of other options were designed into the test chip.

Results. The oscilloscope traces for a worst-case output from the Control Sequencer are shown in Figure V-5. These traces show the Control Sequencer operating at 50 MHz for the non-overlapping clock (left), and 60 MHz for the phi 1-phi 2 overlapping clock signals (right). The output waveforms are shown above the clock waveforms. The top output waveform is the output of a set/reset flip-flop (inverted) whose reset input is propagated through the worst-case PLA path. The lower output waveform is the bit_bottom signal (inverted) which goes high each time the ring counter bit is passed back to the first MSFF in the chain.

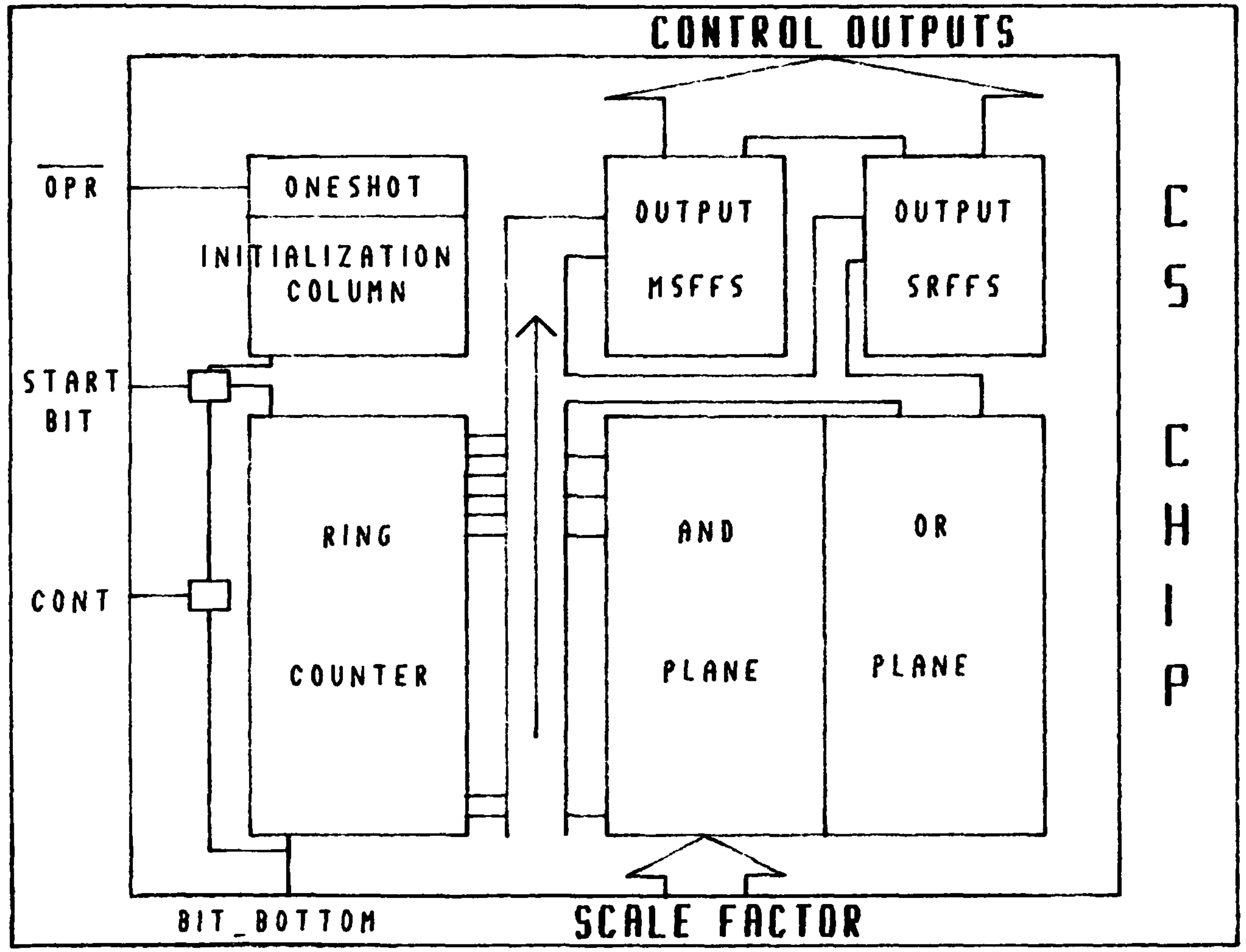


Figure V-4 Control Sequencer Test Chip

The divisions for the clock waveforms of Figure V-5 are 5 nS in duration. The divisions for the Control Sequencer output and bit_bottom signal are 200 nS. Notice that there are 32 clock cycles between pulses for the output signal and bit_ bottom signal.

The available signal generator could only produce clock waveforms of up to 50 MHz. Thus, in order test the circuit at higher speeds, the time between the rise of phi 1 and the fall of phi 2 was decreased. The time between the rise of phi 2 and the fall of phi 1 can be extended without biasing the results since that portion of the clock is only used to propagate the signal into small phi 2 latches, not to drive the signal out to the PLA or chip. By decreasing the separation of the rise of phi 1 to the fall of phi 2 down to 16 nS, a 60 MHz clock rate was approximated. Higher clock frequencies could not be achieved since the overlap of the two-phase clock signals became unacceptable above 60 MHz.

The power consumption of the chip was 6.6 mW for static power dissipation, and 39.7 mW for dynamic power dissipation at 50 MHz. A portion of both power measurements is attributable to a design error that resulted in the shorting of two signal nodes and the floating a large inverter gate.

Evaluation of the Control Circuitry

Control Sequencer. The Control Sequencer test chip demonstrated that the arithmetic/on-chip control logic operates properly at speeds in excess of 60 MHz. These

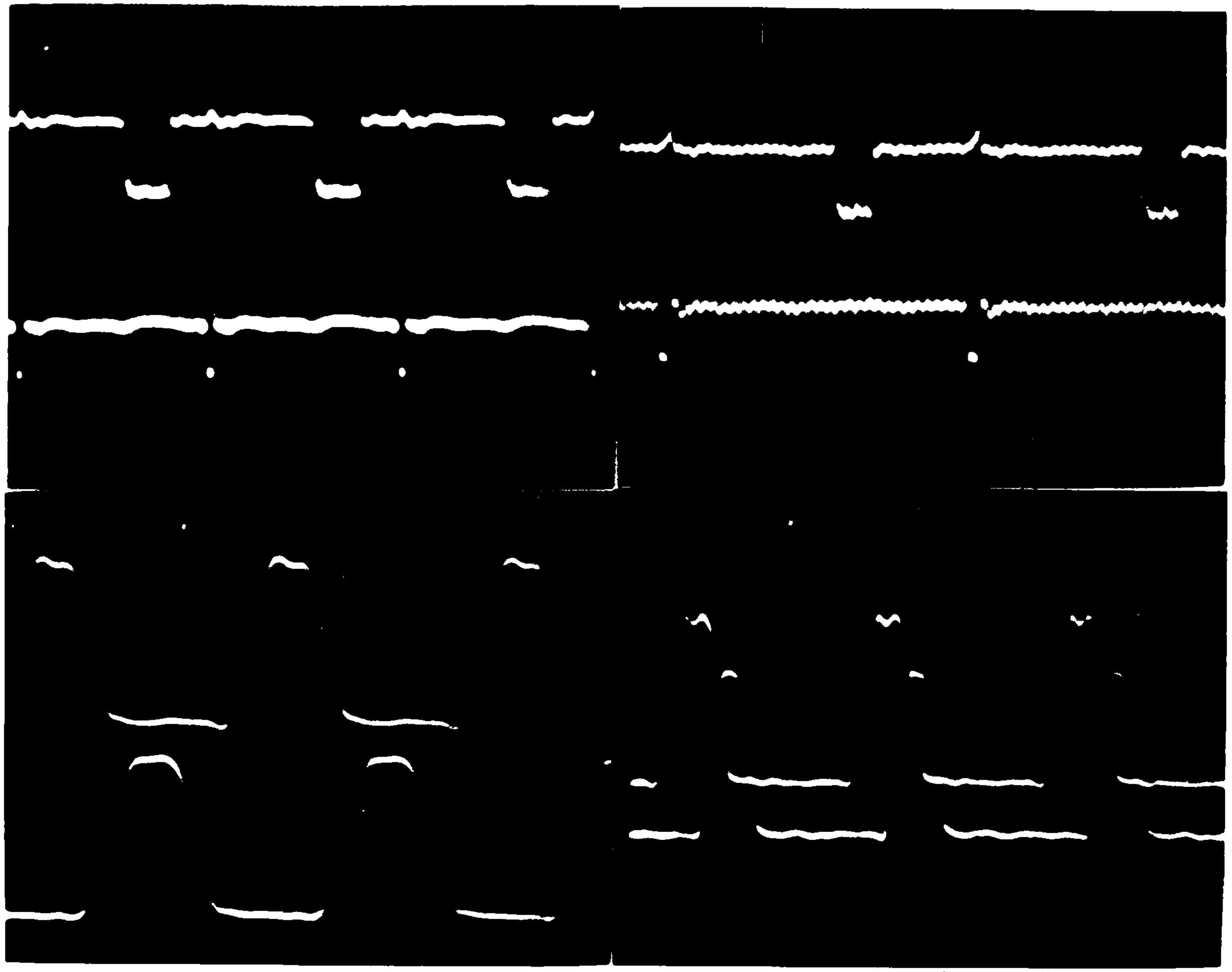


Figure V-5 Control Sequencer Test Results

results exceed the design goals of the 3 micron implementation of the Control Sequencer. The Control Sequencer's ability to operate at the demonstrated clock speeds combined with its low power consumption (under 40 mW) and small layout area (1000 x 2400 lambda) make it an exceptional solution to the arithmetic control requirements. In addition, the Control Sequencer can be easily redesigned if necessary by simply repersonalizing the circuit's PLA. The Control Sequencer's MSFFs that include the Scan-in/Scan-out test circuitry will eventually enable Automatic Test Generation Equipment (ATGE) to easily determine which WFTA processors are fully functional and which have fabrication errors.

<u>XROM Address Generator</u>. The XROM Address Generator circuit is more difficult to evaluate since the test chip was not received in time to test its speed and functionality. However, the circuit's design and SPICE simulations show that it should function properly at clock speeds over 55 MHz for the 3 micron process. The XROM Address Generator circuit shares the same ease of redesign and testability features as the Control Sequencer since the contents of the XROM can be automatically repersonalized if necessary, and since the XROM's MSFFs contain the same type test circuitry as the Control Sequencer MSFFs. Finally, the XROM Address Generator design fits in the available chip area as discussed in Chapter III.

VI. Conclusions and Recommendations

Conclusions

In this thesis, two circuits to control the arithmetic and address generation circuitry of a high performance VLSI WFTA processor at speeds over 50 MHz were designed, simulated, and implemented. A Control Sequencer/PLA circuit was developed to control the on-chip circuitry, and an XROM Address Generator circuit was developed to produce the proper sequence of I/O addresses. Both circuit layouts were designed and submitted for fabrication in 3 micron CMOS. Only the Control Sequencer test chip was received in time to be tested. Test results obtained for the Control Sequencer show proper functionality at clock speeds of over 60 MHz. Although, the XROM Address Generator circuit was not tested, SPICE simulations show that this circuit can operate at clock speeds in excess of 55 MHz. Both circuit designs demonstrated the potential to successfully control the WFTA processors' arithmetic and address generation circuitry in terms of functionality, area, speed, and power.

The control circuitry for the WFTA processors was also shown to be a viable solution in terms of the off-chip interface requirements. An Interface Chip was proposed to coordinate the operation of the WFTA processors in calculating the DFT assigned by a host processor. A general discussion of the control signal requirements and implementation advantages of this approach have been outlined.

A software package that produces an automatic layout of an optimized XROM Address Generator circuit was designed, coded, tested, and utilized. The optimization software uses row and column sign bits and a graph partitioning algorithm to reduce the number of transistors in the XROM. The software then reduces the number of drains in the XROM by reordering its rows and columns in a manner determined by an algorithm that provides a near-optimal solution to row and column Traveling Salesman Problems. The reduction in the number of transistors and drains within the XROM results in an improvement in the XROM's speed, yield, and power consumption. The automatic personalization and layout of the XROM helps to ensure the correctness of the final design.

The realization of the WFTA control circuit designs has its primary importance in the contribution it makes toward achieving the much greater goal of implementing a PFA processor capable of computing 4080-point DFTs at a rate of over 8300 Hz (Taylor, 1985).

Recommendations

The following recommendations are proposed regarding future action:

1. Further testing and modification of the control circuits for the WFTA16 should be pursued. More

extensive test results for both the Control Sequencer and XROM Address Generator circuits are required before a complete WFTA16 processor is submitted for fabrication in the 1.2 micron process. As the control circuit testing and necessary modifications are pursued, continued verification of the WFTA16 timing diagram and module interfaces should be accomplished.

2. The complete timing diagrams for the WFTA15 and WFTA17 processors must be developed, and the control circuits constructed from them using the cells presented in this thesis. Implementation of the WFTA15 and WFTA17 should involve relatively minor revisions to the Control Sequencer and XROM Address Generator once the timing diagram is finalized.

3. The requirements for the Interface chip must be fully investigated and circuit functions fully defined. The Interface chip must then be designed and fabricated in order for the total PFA processing system to be complete.

4. The software that produces an automatic layout of an optimized XROM should be validated and generalized. A menu-driven, front-end software module should be added to allow a user to choose the size and layout configuration of the XROM at run time. In addition to the front-end module, some modifications to the existing software are required such as the use of dynamic memory allocation and reformatting some of the data structures that rely on the XROM using a particular sized multiplexer (4 to 1) on top of the XROM array and between the MSFF banks and address buses.

Other, more general, recommendations for future action regarding the realization of the WFTA16 and PFA processors are given in Taylor (Taylor, 1985: Chapter 6).

Bibliography

Arnold, Michael. "Lyra Design Rule Checker," The Unix Programmer's Manual, 4 University of California at Berkely, 1983.

Baker, Clark. Stat. Personal Correspondance. Massachusetts Institute of Technology, Cambridge, MA; 1 June 1985.

Bellmore, M. and Nemhauser, G., "The Traveling Salesman Problem: A Survey," Operations Research, 16(3): 538-558 (1968).

Blanken, J. D. and P. L. Rustan, "Selection Criteria for Efficient Implementation of FFT Algorithms," IEEE Transactions on Acoustics, Speech, and Signal Processing, 30 (1): 107-109 (February 1982).

Burkard, R. E., "Traveling Salesman and Assignment Problems: A Survey," Annals of Discrete Mathematics, 4: 193-215 (1979).

Burrus, Charles S. and Peter W. Eschenbacher, "An In-Place, In-Order Prime Factor FFT Algorithm," IEEE Transactions on Acoustics, Speech, and Signal Processing, 29 (4): 806-817 (August 1981).

Christofides, N., "Hamiltonian Circuits and the Traveling Salesman Problem: A Survey," in Combinatorial Programming: Methods and Applications, B. Roy, Ed., Reidel, Dordrecht, Holland, 1975.

Cohen, P. B., "An Introduction to CMOS Design Styles," VLSI Design: 88-96 (September 1984).

Collins, Capt James. Simulation and Modeling of a VLSI Winograd Fourier Transform Processor. MS Thesis ENG/GE/85D-9. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1985.

Cooley, J. W., and J. W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series," Math. Comp., 19: 297-301 (1965).

Coutee, Capt Paul. Arithmetic Circuitry for High Speed VLSI Winograd Fourier Transform Algorithm Processors. MS Thesis ENG/GE/85D-11. School of Engineering Air Force Institute of Technology, Wright-Patterson AFB OH, December 1985.

Crowder, H. and M. W. Padberg. "Solving Large Scale Symmetric Traveling Salesman Problems to Optimality." Manage Sci. 26: 495-509 (1980).

Dantzig, G., Fulkerson, D., and S. Johnson, "On a Linear Programming Combinatorial Approach to the Traveling Salesman Problem," Operations Research, 7 (1): 58-66 (1959).

Dionne, R. and M. Florian, "Exact and Approximate Algorithms for Optimal Network Design," Networks, 9 (1): 214-227 (Spr 1979).

Fitzpatrick, Dan, "Mextra Circuit Extractor," The Unix Programmer's Manual, 4. University of California at Berkely, 1983.

Good, I. J., "The Interaction Algorithm and Practical Fourier Anlayisis," Journal of Royal Statistic Society, 20: 361-372 (1958).

Gomory, R. E., "Outline of an Algorithm for Integer Solutions to Linear Programs," Bull. Amer. Math. Soc. 64 275-278 (1958).

Gomory, R. E., "Solving Linear Programming Problems in Integers," Proc. Symp. Appl. Math. 10: 211-215 (1960).

Gomory, R. E., "An Algorithm for Integer Solutions to Linear Programs," Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe, Eds., McGraw-Hill Book Co., Inc., pp. 269-302 New York 1963.

Grotschel, M., and M. W. Padberg, "On the Symmetric Traveling Salesman Problem II: Lifting Theorems and Facets," Mathematical programming, 16: 281-302 (1979).

Hayes, John P. and Edward J. McCluskey, "Testability Considerations in Microprocessor-Based Design," Computer, 3: 17-24 (Mar 1980).

Held, M., and R. M. Karp, "A Dynamic Programming Approach to Sequencing Problems," SIAM, 10: 196-210 (1962).

Held, M., and R. M. Karp, "The Traveling Salesman problem and Minimum Spanning Trees," Operations Research, 18 (6) 1138-1162 (1970).

Held, M. et al. "Aspects of the Traveling Salesman Problem," IBM Journal of Research and Development, 28 (4): 476-486 (July 1984).

Hyafil, L. and R. L. Rivest, "Graph Partioning and Constructing Optimal Decision Trees are Polynomial Complete Problems," Rep. 33, IRIA-Laboria, 1973.

Kernighan, B. W. and C. Ritchie, The C Programming Language. Englewood Cliffs N.J.: Prentice-Hall, 1978.

Kernighan, B. W. and S. Lin, "An Efficient Heuristic Procedure for Paritioning Graphs," The Bell System Technical Journal, 49: 291-307 (February, 1970).

Lewis, E. CMOS Custom Circuit Design. 1983.

Lin, S., and B. W. Kernighan, "An Effective Heuristic Algorithm for the Traveling Salesman problem," Operations Research, 21 498-516 (1973).

Linderman, Richard W. High Performance VLSI Technologies, Integrated Circuits and Architectures for Digital Signal Processing. PhD Thesis. Cornell University, Ithaca, NY, August 1984.

Linderman, Richard, W. et al. "CUSP:A 2-micron CMOS Digital Signal Processor," IEEE Journal of Solid-State Circuits, 20(3): 761-769 (June, 1985).

Little, J. D. et al. "An Algorithm for the Traveling Salesman Problem," Operations Research, 11: 972-984 (1963).

Mano, M. Morris. Digital Logic and Computer Design. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

McClellan, James H. and Charles M. Rader. Number Theory in Digital Signal Processing. Englewood Cliffs, N.J., Prenctice-Hall,1979.

McKenny, V., "A 5V 65K EPROM Utilizing Redundant Circuitry," IEEE International Solid-State Circuits Conference: 146-147, 1980.

Mead, C., and L. Conway, Introduction to VLSI Systems. Reading, MA: Addison-Wesley, 1980.

Nagel, L. et al., " SPICE", The Unix Programmer's Manual, 4. University of California at Berkely, 1983.

Nussbaumer, H. J. Fast Fourier Transform and Convolution Algorithms. Berlin,New York: Springer-Verlag, 1982.

Ong, D. G. Modern Mos Technology: Processes, Devices, and Design. New York, New York: McGraw-Hill, 1984.

Ousterhout, J., "Caesar", The Unix Programmer's Manual, 4. University of California at Berkeley, 1983.

Parker, R. G. and R. L. Rardin, "The Traveling Salesman Problem: An Update of Research," Naval Research Logistic Quarterly, 30: 69-96 (1983).

Rader, Charles M. "Discrete Fourier Transforms When the Number of Data Samples is Prime," Proceedings of the IEEE. 56 (6): 1107-1108 (June 1968).

Taylor, Capt Kent. Architecture and Numerical Accuracy of High-Speed DFT Processing Systems. MS Thesis ENG/GE/85-D-47. School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1985.

Weste, N. and K. Eshraghian, Principles of CMOS VLSI Design: A Systems Perspective. Reading, MA: Addison-Wesley, 1985.

Wilson, D. R. and P. R. Schroeder, "A 100nS 150 mW 64K bit ROM'" IEEE International Solid-State Circuits Conference: 152-153, 1978.

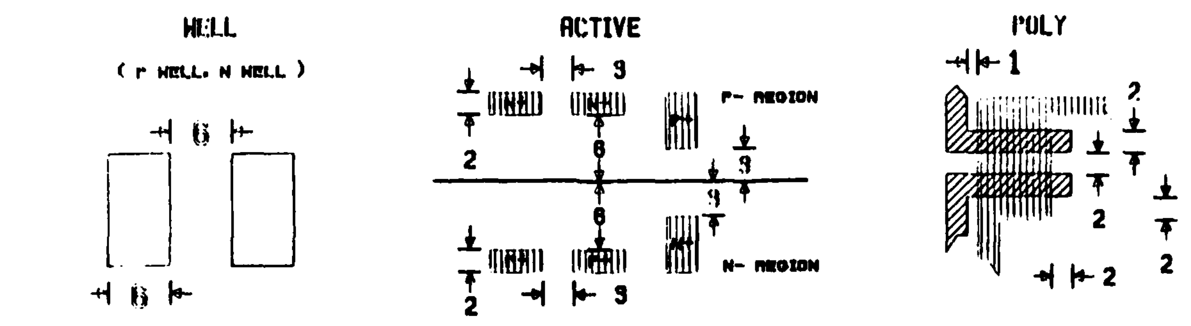
Winograd, Samuel. "On Computing the Discrete Fourier Transform," Mathematics of Computation, 32 (141): 175-199 (January 1978).

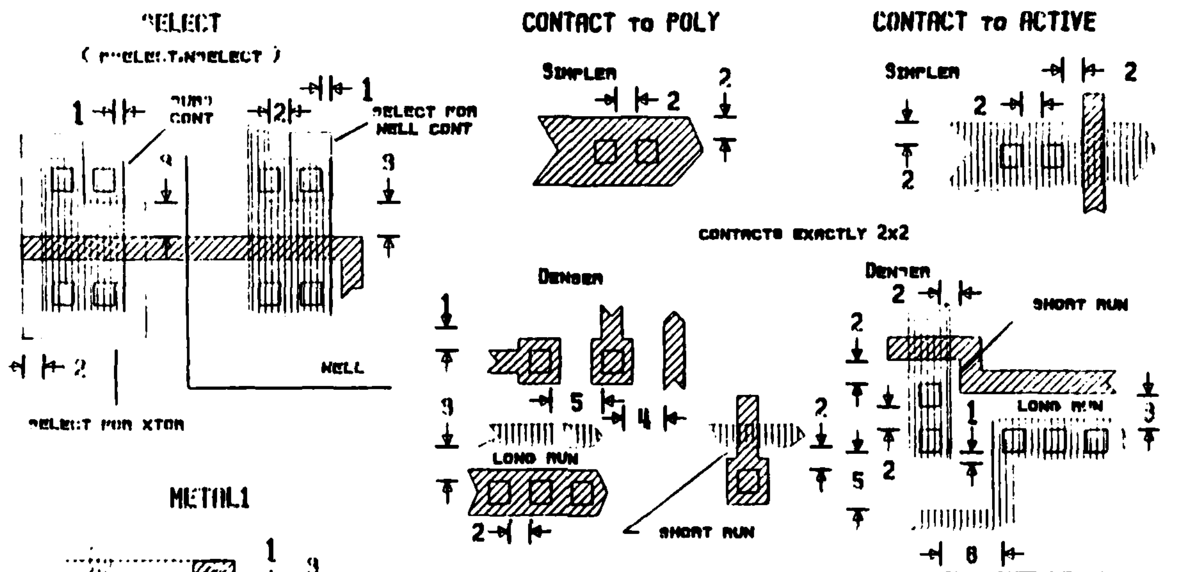
Vita

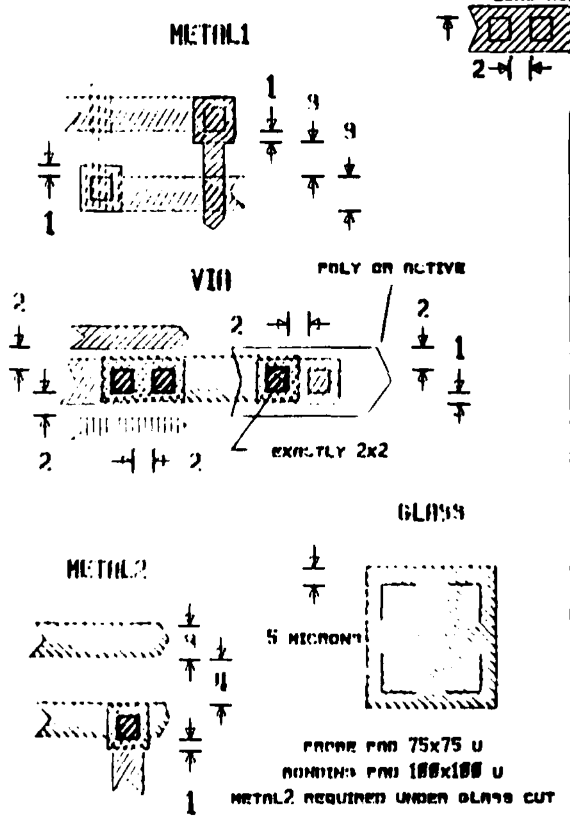
Captain Paul C. Rossbach was born on 27 May 1958 in St. Paul, Minnesota. He graduated from West Point in 1980, and was commissioned a Second Lieutenant in the United States Army, Signal Corps. He served a three-year tour with the 9th Infantry Division at Fort Lewis, Washington before entering the School of Engineering, Air Force Institute of Technology, in July 1984.

Captain Rossbach is a member of Phi Kappa Phi, Tau Beta Pi, and Eta Kappa Nu.

MOSIS CMOS SCALABLE RULES (REV2)







LAYER	CIF	CALMA •	COLOR
WELL	CWG	53	▭
PWELL	CWP	41	▭
NWELL	CWN	42	▭
ACTIVE	CAA	43	▥
SELECT	CSG	54	▭
PSELECT	CSP	44	▭
NSELECT	CSN	45	
POLY	CPG	46	▨
CONT to POLY	CCP	47	▭
CONT to ACT	CCA	48	▭
METAL1	CMF	49	▨
VIA	CVA	50	▨
METAL2	CMS	51	▨
GLASS	COG	52	▭

ALL LAYERS EXCEPT METALS MUST BE ON A LAMBDA GRID
METALS MUST BE ON A HALF LAMBDA GRID

APPENDIX B "PLACEMENT" CODE

```
/*********************************************************************
*                                                                   *
*                                                                   *
*       DATE: 1 DEC 1985                                            *
*       VERSION: 1.0                                                *
*                                                                   *
*       TITLE: PLACE/DEVICE MINIMIZE ROUTINE                        *
*       FILENAME: PLACEMENT.C                                       *
*       COORDINATOR: CPT R W LINDERMAN                              *
*       PROJECT: XROM OPTIMIZER                                     *
*       OPERATING SYSTEM: UNIX V 4.2                                *
*       LANGUAGE: C                                                 *
*       USE: included in gen_XROM.c                                 *
*       CONTENTS:                                                   *
*                       addr_gen()                                  *
*                       exp()                                       *
*                       placement()                                 *
*                       column()                                    *
*                       build_cor_matrix()                          *
*                       col_swap()                                  *
*                       word()                                      *
*                       putaddresses()                              *
*                       array_count()                               *
*                       word_byte_count()                           *
*                       flipbyte()                                  *
*                       setwordbit()                                *
*                       flipword()                                  *
*                       determine_costs()                           *
*                       movcol()                                    *
*                       compare_save()                              *
*                       partition()                                 *
*                       k_and_l()                                   *
*                       calc_D()                                    *
*                       Gmax()                                      *
*                       build_garray()                              *
*                       high_gn()                                   *
*                       Recalc_D()                                  *
*                                                                   *
*       FUNCTION: This program places bits in an XROM and attempts  *
*                 to minimize the total number of transistors in    *
*                 the XROM array by using sign bits and applying    *
*                 the Kernighan & Lin Graph Partitioning Algorithm  *
*                 (Kernighan,B W and S Lin, "An Efficient Heuristic *
*                 Procedure for Partitioning Graphs",Bell System    *
*                 Tecnical Journal,49: 291-308 (1970).              *
*                 The results are passed to the DRAINS.C program    *
*                 for drain minimization.                           *
*                                                                   *
*********************************************************************/
```

```
/*
#include "stdio.h"

#define COLS                           48
#define ROWS                          144
#define GROUPS                          4
#define DATAWIDTH                      12
#define OUT_SIZE        ROWS*GROUPS*DATAWIDTH
*/

#define IN_SIZE              ROWS*GROUPS*8
#define KR_L_TIMES                      3
#define KR_L                            1

#define FIRST                           0
#define SECOND                          1

#define BIG_NUMBER              100000000
char out_array[];
char word_sign_bit[];
int col_sign_bit[];
int carray[];

int costs[COLS][COLS];
int AOones[COLS][ROWS];
int AONones[COLS][ROWS];
int BigD[2][8];
int D[COLS];
int g[COLS/2];
unsigned in_array[IN_SIZE];
int signs[COLS];
char hold[COLS][ROWS];
int save48[COLS];
int save1_24[COLS/2];
int save2_24[COLS/2];
int last48E;
int last1_24E;
int last2_24E;




/***************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME:  ADDR_GEN                                                   *
*       DESCRIPTION:                                                      *
*       This module reads in the data to be placed in the XROM,          *
*       and tries every possible addressing scheme in order to           *
*       minimize the number of ones in the array.  After a               *
*       particular addressing scheme is applied, two methods of applying*
*       the sign bits are performed for each address placement.         *
*       The first method performs the following in the given order:     *
*         the column sign bits are set,                                  *
*         the Kernighan and Lin graph partitioning algorithm             *
*             is applied to the column groups,                           *
*         and then the row sign bits are applied.                        *
*       The second method performs the same functions in a different    *
*       order as follows:                                                *
*         the Kernighan and Lin graph partitioning algorithm             *
*             is applied to the column groups,                           *
*         the row sign bits are applied,                                 *
*         and then the column sign bits are set,                         *
*                                                                        *
*       The addressing scheme and column arrangement, and method         *
*       that yields a minimum number of ones is used                     *
*       for the next step (L_and_K TSP algorithm) and saved.             *
*                                                                        *
*       PASSED VARIABLES: NONE                                           *
*       RETURNS: NONE                                                    *
*       GLOBAL VARIABLES USED: in_array, carray, out_array               *
*       GLOBAL VARIABLES CHANGED:  in_array, carray                      *
*       FILES READ: ADDRESSES                                            *
*       FILES WRITTEN: scoutput                                          *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: exp, placement, column, col_swap, word           *
*                       array_count,build_cor_matrix                     *
*       CALLING MODULES: gen_XROM                                        *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/


addr_gen()

{
  int totalones;
  int i;
  int leastones;
  int a,b,c,d;
  int aa,bb,cc,dd;
  int sA,sB,sC,sD,sE,sF,sG;
  int index;
```

```
        int method;
        int pla[4];
        int scarray[COLS];
        char out_save[OUT_SIZE];


        FILE *fp, *fopen();

        fp=fopen("ADDRESSES","r");

        for (i=0;i<=IN_SIZE-1;i++)
          fscanf(fp,"%u",&in_array[i]);
        fclose(fp);

        leastones = BIG_NUMBER;

        for (aa=2;aa<=8;aa++)
        {
         a=exp(2,aa);
         for (bb=2;bb<=8;bb++)
         {
          b=exp(2,bb);
          if (a!=b)
           for (cc=2;cc<=8;cc++)
           {
            index=0;
            c=exp(2,cc);
            if ((a!=c)&&(b<c))
             {
              for (dd=2;dd<=8;dd++)
              {
               d=exp(2,dd);
               if ((a!=d)&&(b!=d)&&(c!=d))
                 pla[index++]=d;
              }
              placement(a,b,c,pla[0],pla[1],pla[2],pla[3]);
              for (i=0;i<OUT_SIZE;i++)
                out_save[i] = out_array[i];
              column();

#ifdef KR_L
                build_cor_matrix();
                col_swap(0);
#endif

              word();
              totalones = array_count();
              if (totalones < leastones)
               {
                 method = FIRST;
                 leastones = totalones;
                 sA=a;
                sB=b;
                sC=c;
                sD=pla[0];
                sE=pla[1];
                sF=pla[2];
                sG=pla[3];

                for (i=0;i<=COLS-1;i++)
                  scarray[i]=carray[i];
              }


            for (i=0;i<OUT_SIZE;i++)
              out_array[i] = out_save[i];

    #ifdef KR_L
               build_cor_matrix();
               col_swap(0);
    #endif
            word();
            column();
            totalones = array_count();

            if (totalones < leastones)
             {
                method = SECOND;
                leastones = totalones;
                sA=a;
                sB=b;
                sC=c;
                sD=pla[0];
                sE=pla[1];
                sF=pla[2];
                sG=pla[3];

                for (i=0;i<=COLS-1;i++)
                  scarray[i]=carray[i];
              }
            }
          }
        }
      }

      placement(sA,sB,sC,sD,sE,sF,sG);
      if ( method == FIRST )
        {
          column();
          for (i=0;i<=COLS-1;i++)
            carray[i]=scarray[i];

    #ifdef KR_L
        col_swap(1);
```

```
#endif

      word();
    }
  else
    {
      for (i=0;i<=COLS-1;i++)
        carray[i]=scarray[i];

#ifdef KR_L
    col_swap(1);
#endif

      word();
      column();
    }


 fp=fopen("scoutput","w");

 fprintf(fp,"address_A %d\n",sA);
 fprintf(fp,"address_A %d\n",sB);
 fprintf(fp,"address_A %d\n",sC);
 fprintf(fp,"address_A %d\n",sD);
 fprintf(fp,"address_A %d\n",sE);
 fprintf(fp,"address_A %d\n",sF);
 fprintf(fp,"address_A %d\n",sG);

 fclose(fp);

 return;

}



/************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME:  EXP                                                      *
*       DESCRIPTION:                                                    *
*       This module calculates the exponent of x raised to the          *
*       y power.                                                        *
*                                                                       *
*       PASSED VARIABLES: x: argument                                   *
*                         y: exponent                                   *
*       RETURNS:  x ** y                                                *
*       GLOBAL VARIABLES USED: NONE                                     *
*       GLOBAL VARIABLES CHANGED:  NONE                                      *
*       FILES READ:                                                          *
*       FILES WRITTEN:                                                       *
*       HARDWARE INPUT:                                                      *
*       HARDWARE OUTPUT:                                                     *
*       MODULES CALLED: NONE                                                 *
*       CALLING MODULES: addr_gen                                            *
*                                                                            *
*       AUTHOR: PAUL ROSSBACH                                                *
*       HISTORY:                                                             *
*                                                                            *
*                                                                            *
****************************************************************************/



exp(x,y)
int x;
int y;

{

 int total;
 int i;


  total=x;
  if (y<=1)
    {
     printf("IN EXP exponent less than 2 - ABORT\n");
     exit();
    }

  else
    for (i=2;i<=y;i++)
       total=total*x;


 return(total);
}



/****************************************************************************
*                                                                            *
*                                                                            *
*       DATE: 1 DEC 1985                                                     *
*       VERSION: 1.0                                                         *
*                                                                            *
*       NAME: PLACEMENT                                                      *
*       DESCRIPTION:                                                         *
*       The Placement module receives an addressing scheme for the        *
*       XROM data and places each data bit in the correct location        *
*       in the XROM for that particular addressing scheme.                *
*                                                                         *
*       PASSED VARIABLES:     a_i: increment for XROM address             *
*                         a_mux1: increment for low mux address           *
*                         a_mux1: increment for high mux address          *
*                         a_pla1: increment for pla decoder address 1     *
*                         a_pla2: increment for pla decoder address 2     *
*                         a_pla3: increment for pla decoder address 3     *
*                         a_pla4: increment for pla decoder address 4     *
*                                                                         *
*       RETURNS: NONE                                                     *
*       GLOBAL VARIABLES USED: in_array, out_array                        *
*       GLOBAL VARIABLES CHANGED: out_array                               *
*       FILES READ:                                                       *
*       FILES WRITTEN:                                                    *
*       HARDWARE INPUT:                                                   *
*       HARDWARE OUTPUT:                                                  *
*       MODULES CALLED: putaddresses                                      *
*       CALLING MODULES: addr_gen                                         *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
**************************************************************************/


placement(a_i,a_mux1,a_mux2,a_pla1,a_pla2,a_pla3,a_pla4)
int a_i,a_mux1,a_mux2,a_pla1,a_pla2,a_pla3,a_pla4;
{

  int word;
  int addr1,addr2,addr3,addr4,addr5,addr6,addr7,addr8;
  int baseaddr;
  unsigned current_addr[8];
  int row_count;
  int i;
  int baseai;
  int mux1_2;


    for (i=0;i<=OUT_SIZE-1;i++)
        out_array[i]=0;


    mux1_2=a_mux1+a_mux2;

    row_count=0;
    for (addr8=0;addr8<=4096;addr8 += 4096)
     for (addr7=addr8;addr7<=addr8+2048;addr7 +=2048)
```

```
          for (addr6=addr7;addr6<=addr7+1024;addr6 += 1024)
            for (addr5=addr6;addr5<=addr6+512;addr5 += 512)
             if (addr8==0 || ((addr8==4096) && (addr5==4096) ))
             {
             for (addr4=addr5;addr4<=addr5+a_pla4;addr4 += a_pla4)
              for (addr3=addr4;addr3<=addr4+a_pla3;addr3 += a_pla3)
               for (addr2=addr3;addr2<=addr3+a_pla2;addr2 += a_pla2)
                for (addr1=addr2;addr1<=addr2+a_pla1;addr1 += a_pla1)
                {
                 for (baseaddr=addr1;baseaddr<=(addr1+3);baseaddr++)
                 {
                  baseai=baseaddr+a_i;
                  current_addr[0]=in_array[baseaddr];
                  current_addr[1]=in_array[baseai];
                  current_addr[2]=in_array[baseai+a_mux1];
                  current_addr[3]=in_array[baseaddr+a_mux1];
                  current_addr[4]=in_array[baseaddr+a_mux2];
                  current_addr[5]=in_array[baseai+a_mux2];
                  current_addr[6]=in_array[baseai+mux1_2];
                  current_addr[7]=in_array[baseaddr+mux1_2];


                  word=baseaddr-addr1;


                  putaddresses(current_addr,row_count,word);
                 }
                 row_count++;

              }
          }
     return;

    }




    /*************************************************************************
    *                                                                       *
    *                                                                       *
    *       DATE: 1 DEC 1985                                                *
    *       VERSION: 1.0                                                    *
    *                                                                       *
    *       NAME: COLUMN                                                    *
    *       DESCRIPTION:                                                    *
    *       The column module determines if the column should be            *
    *       inverted and the corresponding column sign bit set.             *
    *       It does this by counting all the one-bits in the column         *
    *       and comparing that total with half the total number of          *
    *       bits in the column.  If the number of ones is over half         *
```

```
*       (+1), the column is inverted and sign bit set.  Column          *
*       also calculates a "number of one-bits state" for each           *
*       AO/AO-not byte half that is used later to easily calculate      *
*       the distance between two columns.                               *
*                                                                       *
*       PASSED VARIABLES: NONE                                          *
*       RETURNS: NONE                                                   *
*       GLOBAL VARIABLES USED: col_sign_bit                             *
*       GLOBAL VARIABLES CHANGED:  col_sign_bit                         *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: bytecount, flipbyte                             *
*       CALLING MODULES: addr_gen                                       *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
***********************************************************************/


column()
{

 int count;
 int totalcount;
 int i,j;


 count=0;
 totalcount=0;


 for (i=0;i<=COLS-1;i++)
 {
  totalcount=0;
  col_sign_bit[i]=0;
  for (j=0;j<=ROWS-1;j++)
   {
    count=bytecount(i+COLS*j);
    totalcount += count;
   }
  if (totalcount>=(ROWS*4+1))
   {
    for (j=0;j<=ROWS-1;j++)
      flipbyte(i+COLS*j);
    col_sign_bit[i]=1;
   }
  }
 return;
}




/***********************************************************************
*                                                                      *
*                                                                      *
*       DATE: 1 DEC 1985                                               *
*       VERSION: 1.0                                                   *
*                                                                      *
*       NAME: BUILD_COR_MATRIX                                         *
*       DESCRIPTION:                                                   *
*       The build_cor_matrix module calculates a                       *
*               "number of one-bits state"                             *
*       for each AO/AO-not byte half that is used later to easily      *
*       calculate the distance between two columns.                    *
*                                                                      *
*       PASSED VARIABLES: NONE                                         *
*       RETURNS: NONE                                                  *
*       GLOBAL VARIABLES USED:  AOones, AONones                        *
*       GLOBAL VARIABLES CHANGED:  AOones, AONones                     *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED: word_byte_count                                *
*       CALLING MODULES: addr_gen                                      *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/


build_cor_matrix()
{

 int i,j;
 int AOcount,AONcount;


 for (i=0;i<=COLS-1;i++)
 {
  for (j=0;j<=ROWS-1;j++)
  {
    AOcount=word_byte_count(i+COLS*j,0);
    AOones[i][j]=AOcount-2;
    AONcount=word_byte_count(i+COLS*j,1);
    AONones[i][j]=AONcount-2;
  }
 }
 return;

}


/***************************************************************************
*                                                                          *
*                                                                          *
*       DATE: 1 DEC 1985                                                   *
*       VERSION: 1.0                                                       *
*                                                                          *
*       NAME: COL_SWAP                                                     *
*       DESCRIPTION:                                                       *
*       The Col_swap module is the highest level of the Kernighan          *
*       and Lin (K_and_L) graph partitioning algorithm.  It                *
*       calls the routine that calculates the cost-distance                *
*       correlation matrix for the columns, randomly partitions the        *
*       columns, calls the K_and_L algorithm, saves the best result,       *
*       and finally moves the columns to the location indicated            *
*       by the best result.                                                *
*                                                                          *
*       PASSED VARIABLES: final: 0: normal operation                       *
*                                1: puts xrom bits in best pattern         *
*                                   found (for addressing & k_and_l)       *
*       RETURNS: NONE                                                      *
*       GLOBAL VARIABLES USED: carray                                      *
*       GLOBAL VARIABLES CHANGED:  carray                                  *
*       FILES READ:                                                        *
*       FILES WRITTEN:                                                     *
*       HARDWARE INPUT:                                                    *
*       HARDWARE OUTPUT:                                                   *
*       MODULES CALLED: determine_costs, partition, k_and_l                *
*                       compare_save, movcol                               *
*       CALLING MODULES: addr_gen                                          *
*                                                                          *
*       AUTHOR: PAUL ROSSBACH                                              *
*       HISTORY:                                                           *
*                                                                          *
*                                                                          *
***************************************************************************/


col_swap(final)
```

```
int final;
{

  int choices[COLS];
  int pulled[COLS];
  int filetemp[COLS];
  int temp;
  int i,j;
  int x,y;


  if (final==0)
  {
  srandom(1);

  determine_costs();
  for (i=0;i<=COLS-1;i++)
    choices[i]=i;

  for (i=0;i<=KR_L_TIMES-1;i++)
  {
    partition(COLS,choices);
    k_and_l(COLS);
    if (i==0)
      compare_save(COLS,0,0);
    else
      compare_save(COLS,3,0);
  }
  for (i=0;i<=COLS-1;i++)
    choices[i]=carray[i];
  for (i=0;i<=KR_L_TIMES-1;i++)
  {
    partition(COLS/2,choices);
    k_and_l(COLS/2);
    compare_save(COLS/2,1,0);
  }
  for (i=0;i<=COLS/2-1;i++)
    choices[i]=choices[i+COLS/2];

  for (i=0;i<=KR_L_TIMES-1;i++)
  {
    partition(COLS/2,choices);
    k_and_l(COLS/2);

    if (i==(KR_L_TIMES-1))
      compare_save(COLS/2,2,1);
    else
      compare_save(COLS/2,2,0);
  }

  } /* end final==0 if */
```

```
 for (i=0;i<=((GROUPS-1)*DATAWIDTH);i +=DATAWIDTH)
   for (j=0;j<=DATAWIDTH-1;j++)
     if ((carray[i+j]<=(i+DATAWIDTH-1)) && (carray[i+j]>=i))
       {
         temp=carray[i+j];
         carray[i+j]=carray[temp];
         carray[temp]=temp;
       }
 for (y=0;y<=COLS-1;y++)
   {
     pulled[y]=0;
     filetemp[y]=0;
   }
 temp=(-1);

 for (x=0;x<=COLS-1;x++)
   if (carray[x]!=x)
     {
      if (pulled[x]==0)
        {
          movcol(x,temp);
          filetemp[x]=1;
        }
      if (filetemp[carray[x]]==0)
        {
          movcol(carray[x],x);
          pulled[carray[x]]=1;
        }
      else
         movcol(temp,x);
     }

return;

}


/*********************************************************************
*                                                                    *
*                                                                    *
*       DATE: 1 DEC 1985                                             *
*       VERSION: 1.0                                                 *
*                                                                    *
*       NAME: WORD                                                   *
*       DESCRIPTION:                                                 *
*       The word module performs two functions.  It counts the number  *
*       of one-bits in a row for A0 and A0-not to see which rows      *
*       should be inverted, and it gives the total number of ones in  *
*       the array after all rows have been checked.  The rows are     *
*       inverted if the number of ones is over half as with the columns.*
*       The total number of ones is kept as a running total updated   *
*       after each row is checked.                                    *
*                                                                      *
*       PASSED VARIABLES: NONE                                         *
*       RETURNS:          grandtotal: total ones in array = answer     *
*       GLOBAL VARIABLES USED: word_sign_bit                           *
*       GLOBAL VARIABLES CHANGED:  word_sign_bit                       *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED: word_byte_count, flipword, setwordbit          *
*       CALLING MODULES: addr_gen                                      *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/


word()
{

 int count;
 int totalcount;
 int i,j;
 int word;
 int AON;
 int rowbyte;
 int wordbyte;


  for (i=0;i<=ROWS-1;i++)
  {
   rowbyte=COLS*i;
   word_sign_bit[i]=0;
   for (word=0;word<=3;word++)
    {
    wordbyte=DATAWIDTH*word;
    for (AON=0;AON<=1;AON++)
     {
        totalcount=0;
        for (j=0;j<=DATAWIDTH-1;j++)
       {
        count=word_byte_count((wordbyte+j+rowbyte),AON);
        totalcount += count;
       }
       if (totalcount>=(DATAWIDTH*2+1))
       {
        for (j=0;j<=DATAWIDTH-1;j++)
         flipword((wordbyte+j+rowbyte),AON);
        setwordbit(i,(word*2+AON));
       }
```

```
          }
        }
     }

     return;

   }




/************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME: PUTADDRESSES                                              *
*       DESCRIPTION:                                                    *
*       The putaddresses module performs the placement of the           *
*       bits for a particular row and word (there are 4 words of        *
*       DATAWIDTH per row) from the 8 data entries in address           *
*       array (addrarray).  Each row of the XROM with a 4 to 1          *
*       Demultiplexer on top of the bit lines has eight different       *
*       data values per row if the XROM only output one dataword at a   *
*       time.  Thus, eight data entries are loaded into the row-word.   *
*                                                                       *
*       PASSED VARIABLES: addrarray: the 8 data entries to place        *
*                               row: the current row                    *
*                              word: the group on that row ( 1 to 4)    *
*       RETURNS: NONE                                                   *
*       GLOBAL VARIABLES USED:  out_array                               *
*       GLOBAL VARIABLES CHANGED: out_array                             *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: NONE                                            *
*       CALLING MODULES: placement                                      *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
************************************************************************/


putaddresses(addrarray,row,word)
int row;
int word;
```

```
unsigned addrarray[8];

{

 int inc,i;
 int bit;
 int rowbyte;
 int wordbyte;
 int byte;
 int mask;
 int bitloc;
 unsigned calc;


 wordbyte=word*DATAWIDTH;
 rowbyte=row*COLS;

  for (inc=0;inc<=7;inc++)
   {
   bitloc=7-inc;
   for (i=DATAWIDTH-1;i>=0;i--)
    {
     bit=((addrarray[inc]>>(DATAWIDTH-1-i)) & (0001));
     if (bit != 0)
        {
        byte=rowbyte+wordbyte+i;
        mask=1;
        mask=(mask<<(bitloc));
        calc=out_array[byte];
        calc=(calc & 0377);
        calc=(calc^mask);
        out_array[byte]=calc;
        }
    }

   }

 return;

}


/***************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME: ARRAY_COUNT                                                 *
*       DESCRIPTION:                                                      *
*       This module counts the total number of ones in the out_array     *
*       PASSED VARIABLES: NONE                                              *
*       RETURNS: totalcount                                                 *
*       GLOBAL VARIABLES USED:  NONE                                        *
*       GLOBAL VARIABLES CHANGED:  NONE                                     *
*       FILES READ:  NONE                                                   *
*       FILES WRITTEN:  NONE                                                *
*       HARDWARE INPUT: NONE                                                *
*       HARDWARE OUTPUT:  NONE                                              *
*       MODULES CALLED: bytecount                                           *
*       CALLING MODULES: addr_gen                                           *
*                                                                           *
*       AUTHOR: PAUL ROSSBACH                                               *
*       HISTORY:                                                            *
*                                                                           *
*                                                                           *
*************************************************************************/


array_count()
{

  int count;
  int totalcount;
  int i;

   count=0;
   totalcount=0;
   for (i=0;i<OUT_SIZE;i++)
   {
    count=bytecount(i);
    totalcount += count;
   }

 return(totalcount);

}




/*************************************************************************
*                                                                           *
*                                                                           *
*       DATE: 1 DEC 1985                                                    *
*       VERSION: 1.0                                                        *
*                                                                           *
*       NAME: BYTECOUNT                                                     *
*       DESCRIPTION:                                                        *
*       This module counts the number of ones in a single byte of the      *
*                                                            out_array   *
*       PASSED VARIABLES: byte                                            *
*       RETURNS: count                                                    *
*       GLOBAL VARIABLES USED: out_array                                  *
*       GLOBAL VARIABLES CHANGED:  NONE                                   *
*       FILES READ:  NONE                                                 *
*       FILES WRITTEN: NONE                                               *
*       HARDWARE INPUT: NONE                                              *
*       HARDWARE OUTPUT:  NONE                                            *
*       MODULES CALLED: NONE                                              *
*       CALLING MODULES: array_count                                      *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
**************************************************************************/


bytecount(byte)
int byte;

{

 int count;
 unsigned calc;


  calc=out_array[byte];
  calc=(calc & 0377);
  for (count=0;calc!=0;calc>>=1)
     if (calc & 001)
        count++;

 return(count);

}




/**************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME:  WORD_BYTE_COUNT                                            *
*       DESCRIPTION:                                                      *
*       This module counts the number or one-bits in the AO or AO         *
*       -not half of one byte of the out_array.  The module is used      *
*       for determining if a row (AO/AO-not) should be inverted.          *
*       The masking patterns are as such to align with the actual         *
*       bit locations of the XROM.  The count for a single  byte          *
*       (AO/AO-not) is returned.                                          *
*                                                                         *
*       PASSED VARIABLES: byte: the index for the out_array               *
*                         AOorAON: 0 or 1 - which half of byte            *
*       RETURNS: count of ones in byte half                               *
*       GLOBAL VARIABLES USED: out_array                                  *
*       GLOBAL VARIABLES CHANGED:  out_array                              *
*       FILES READ:                                                       *
*       FILES WRITTEN:                                                    *
*       HARDWARE INPUT:                                                   *
*       HARDWARE OUTPUT:                                                  *
*       MODULES CALLED: NONE                                              *
*       CALLING MODULES: word                                             *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
*************************************************************************/


word_byte_count(byte,AOorAON)
int byte;
int AOorAON;

{

 int count;
 unsigned calc;


  if (AOorAON)
    {
    calc=out_array[byte];
    calc=(calc & 00146);
    }
  else
    {
    calc=out_array[byte];
    calc=(calc & 00231);
    }
  for (count=0;calc!=0;calc>>=1)
       if (calc & 01)
          count++;
 return(count);

}
```

```
/***********************************************************************
*                                                                      *
*                                                                      *
*       DATE: 1 DEC 1985                                               *
*       VERSION: 1.0                                                   *
*                                                                      *
*       NAME: FLIPBYTE                                                 *
*       DESCRIPTION:                                                   *
*       This module simply inverts an entire character byte of         *
*       the out_array.  The byte number that is to be inverted         *
*       is sent to the module.                                         *
*                                                                      *
*       PASSED VARIABLES: byte: index for out_array                    *
*       RETURNS: NONE                                                  *
*       GLOBAL VARIABLES USED: out_array                               *
*       GLOBAL VARIABLES CHANGED:  out_array                           *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED:  NONE                                          *
*       CALLING MODULES: column                                        *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/



flipbyte(byte)
int byte;

{
  unsigned flip;

  flip=out_array[byte];
  flip=(flip & 0377);
  flip=(flip^(0377));
  out_array[byte]=flip;

 return;

}
```

```
/**************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME:  SETWORDBIT                                                 *
*       DESCRIPTION:                                                      *
*       Setwordbit sets the proper word sign bit for the section          *
*       of the XROM that it is sent.  For each row there are 8 word       *
*       sign bits in the 4 to 1 demultiplexed XROM.  The 8 word           *
*       sign bits are stored in one character byte of the word_           *
*       sign_bit array.  Thus, bit level operations are needed to         *
*       set each sign bit.                                                *
*                                                                         *
*       PASSED VARIABLES: byte: word_sign_bit index                       *
*                          bit: which sign bit in the byte                *
*       RETURNS: NONE                                                     *
*       GLOBAL VARIABLES USED: word_sign_bit                              *
*       GLOBAL VARIABLES CHANGED:  word_sign_bit                          *
*       FILES READ:                                                       *
*       FILES WRITTEN:                                                    *
*       HARDWARE INPUT:                                                   *
*       HARDWARE OUTPUT:                                                  *
*       MODULES CALLED:                                                   *
*       CALLING MODULES:  word                                            *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
**************************************************************************/


setwordbit(byte,bit)
int byte;
int bit;

{
 unsigned calc;
 int mask;

  mask=1;
  mask=(mask<<(7-bit));
  calc=word_sign_bit[byte];
  calc=(calc & 0377);
  calc=(calc^mask);
  word_sign_bit[byte]=calc;

 return;

}
```

```
/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: FLIPWORD                                                   *
*       DESCRIPTION:                                                     *
*       Flipword performs the same function as "flipbyte" except         *
*       that it only inverts half of the byte.  The AO bits of a         *
*       byte are inverted if called with a 0 as the second argument.    *
*       It inverts the AO-not bitss if called with a 1 as the second    *
*       argument.                                                        *
*                                                                        *
*       PASSED VARIABLES: byte: out_array index                          *
*                       AOorAON: 0 or 1 - which word of byte             *
*       RETURNS: NONE                                                    *
*       GLOBAL VARIABLES USED: out_array                                 *
*       GLOBAL VARIABLES CHANGED:  out_array                             *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES: word                                            *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/


flipword(byte,AOorAON)
int byte;
int AOorAON;

{
  unsigned flip;
  if (AOorAON==0)
     {
     flip=out_array[byte];
     flip=(flip & 0377);
     flip=(flip^(0231));
     out_array[byte]=flip;
     }
  else
     {
     flip=out_array[byte];
     flip=(flip & 0377);
     flip=(flip^(0146));
     out_array[byte]=flip;
     }

 return;

}




/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: DETERMINE_COSTS                                            *
*       DESCRIPTION:                                                     *
*       Determine_costs calculates the correlation distance              *
*       between all columns of the XROM for one-bit removal.             *
*       The distance matrix is upper-triangular since the distances      *
*       are symmetric.  The lower triangle is filled in also for future *
*       use.  The AO_ones and AON_ones arrays calculated in the column   *
*       module are used to determine the column's correlation.  The      *
*       cost difference is the number of ones that can be removed if     *
*       two columns are placed under the control of a single sign bit.   *
*       The cost distance between two columns for one-bit removals       *
*       is based on the number of one-bits in the AO and AO_not words    *
*       for each row.                                                    *
*                                                                        *
*       PASSED VARIABLES:  NONE                                          *
*       RETURNS:  NONE                                                   *
*       GLOBAL VARIABLES USED: AOones, AONones, costs                    *
*       GLOBAL VARIABLES CHANGED:  costs                                 *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES:  col_swap                                     *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/



determine_costs()

{


 int AOvector;
 int AONvector;
 int i,x,y;
 int AOtotal;
 int AONtotal;



  for (x=0;x<=COLS-1;x++)
    for (y=0;y<=COLS-1;y++)
       if (x>y)
       {
         AOvector=0;
         AONvector=0;
         for (i=0;i<=ROWS-1;i++)
          {
            AOtotal=AOones[x][i] + AOones[y][i];
            AONtotal=AONones[x][i] + AONones[y][i];
            if (AOtotal < 0)
              AOtotal=(-AOtotal);
            if (AONtotal < 0)
              AONtotal=(-AONtotal);
            AOvector=AOvector+AOtotal;
            AONvector=AONvector+AONtotal;
          }

         costs[x][y]=AOvector+AONvector;

       }

   for (x=0;x<=COLS-1;x++)
     for (y=0;y<=COLS-1;y++)
       if (x<y)
         costs[x][y]=costs[y][x];

 return;
```

```
}


/**********************************************************************
*                                                                     *
*                                                                     *
*       DATE: 1 DEC 1985                                              *
*       VERSION: 1.0                                                  *
*                                                                     *
*       NAME: MOVCOL                                                  *
*       DESCRIPTION:                                                  *
*       Movcol moves the columns of the XROM in order to place them   *
*       in the locations determined by the K_and_L graph partitioning *
*       algorithm.  The module can move columns in three ways.        *
*          a.  from one column position in the XROM                   *
*              to another.                                            *
*          b.  from one column postion to a holding area.             *
*          c.  from the holding area to a column position.            *
*                                                                     *
*       PASSED VARIABLES: from: column index number or -1(for temp area)*
*                           to: column index number or -1(for temp area)*
*       RETURNS: NONE                                                 *
*       GLOBAL VARIABLES USED: col_sign_bit, out_array, hold, signs   *
*       GLOBAL VARIABLES CHANGED:  col_sign_bit, out_array, hold, signs *
*       FILES READ:                                                   *
*       FILES WRITTEN:                                                *
*       HARDWARE INPUT:                                               *
*       HARDWARE OUTPUT:                                              *
*       MODULES CALLED: NONE                                          *
*       CALLING MODULES:  col_swap                                    *
*                                                                     *
*       AUTHOR: PAUL ROSSBACH                                         *
*       HISTORY:                                                      *
*                                                                     *
*                                                                     *
**********************************************************************/


movcol(from,to)
int from;
int to;

{

  int index;


   if ((to>=0) && (from>=0))
    {
      col_sign_bit[to]=col_sign_bit[from];
      for (index=0;index<=ROWS-1;index++)
        out_array[to+COLS*index]=out_array[from+COLS*index];
    }

   if (to<0)
    {
      signs[from]=col_sign_bit[from];
      for (index=0;index<=ROWS-1;index++)
        hold[from][index]=out_array[from+COLS*index];
    }

   if (from<0)
    {
      col_sign_bit[to]=signs[carray[to]];
      for (index=0;index<=ROWS-1;index++)
        out_array[to+COLS*index]=hold[carray[to]][index];
    }

 return;

}


/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: COMPARE_SAVE                                               *
*       DESCRIPTION:                                                     *
*       Since the K_and_L algorithm attempts to minimize the             *
*       ones in the XROM by partitioning the columns numerous            *
*       times, a mechanism is needed to keep track of which solution     *
*       was the best.  Compare_save does the for this first cut in       *
*       half of the XROM, and for each half into quarters.  The          *
*       module will also update the carry array with the best solution   *
*       so far when called.                                              *
*                                                                        *
*       PASSED VARIABLES: size: COLS or COLS/2 partition size            *
*                         time: 0,1,or 2 - try number for a size         *
*                         done: flags the last try for the XROM          *
*       RETURNS: NONE                                                    *
*       GLOBAL VARIABLES USED: costs, carray                             *
*       GLOBAL VARIABLES CHANGED:  carray                                *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED:  NONE                                            *
*       CALLING MODULES:  col_swap                                            *
*                                                                             *
*       AUTHOR: PAUL ROSSBACH                                                 *
*       HISTORY:                                                              *
*                                                                             *
*                                                                             *
*****************************************************************************/



compare_save(size,time,done)
int size;
int time;
int done;

{

 int Etotal;
 int x,y,i;
 int link;



 Etotal=0;

 if ( time==0 )
   {
    last48E=BIG_NUMBER;
    last1_24E=BIG_NUMBER;
    last2_24E=BIG_NUMBER;
   }

 for (x=0;x<=(size/2-1);x++)
  for (y=size/2;y<=size-1;y++)
    {
     link=costs[carray[x]][carray[y]];
     Etotal=Etotal+link;
    }

 if (size==COLS)
  {
   if (Etotal<last48E)
     for (i=0;i<=COLS-1;i++)
       save48[i]=carray[i];
   else
     for (i=0;i<=COLS-1;i++)
       carray[i]=save48[i];
  }
 else
  {
  if (time==1)
```

```
    {
    if (Etotal<last1_24E)
      for (i=0;i<=COLS/2-1;i++)
        save1_24[i]=carray[i];
    }
   else
    {
    if (Etotal<last2_24E)
      for (i=0;i<=COLS/2-1;i++)
        save2_24[i]=carray[i];
    }
   }
  if (done==1)
    for (i=0;i<=COLS/2-1;i++)
      {
        carray[i]=save1_24[i];
        carray[i+COLS/2]=save2_24[i];
      }

  return;

}




/***********************************************************************
*                                                                      *
*                                                                      *
*       DATE: 1 DEC 1985                                               *
*       VERSION: 1.0                                                   *
*                                                                      *
*       NAME: PARTITION                                                *
*       DESCRIPTION:                                                   *
*       This module randomly partitions a group of columns into        *
*       two groups of half the "size".  The initial partition          *
*       is used as a random starting point for the K_and_L             *
*       algorithm.  The columns are partitioned through the column     *
*       number's place in the carray array.                            *
*                                                                      *
*       PASSED VARIABLES: size: COLS or COLS/2 partition size          *
*                      choices: the column numbers to partition        *
*       RETURNS: NONE                                                  *
*       GLOBAL VARIABLES USED:  carray                                 *
*       GLOBAL VARIABLES CHANGED:  carray                              *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED:  NONE                                          *
*       CALLING MODULES:  col_swap                                     *
*                                                                      *
*        AUTHOR: PAUL ROSSBACH                                              *
*        HISTORY:                                                           *
*                                                                           *
*                                                                           *
***************************************************************************/


partition(size,choices)
int size;
int choices[COLS];


{

 int i,j,x;
 int value;
 long random();

 i=0;
 x=0;
 j=size/2;

 while ((i<=(size/2-1)) && (j<=size-1))
   {
     value=random();
     value=(value & 01);

     if (value)
       carray[i++]=choices[x];
     else
       carray[j++]=choices[x];
     x++;
   }

 if (x<size)
   if (i<=(size/2-1))
     while (x<=size-1)
         {
          carray[i++]=choices[x];
          x++;
         }
   else
     while (x<=size-1)
         {
          carray[j++]=choices[x];
          x++;
         }

 return;
```

```
}




/***********************************************************************
*                                                                      *
*                                                                      *
*       DATE: 1 DEC 1985                                               *
*       VERSION: 1.0                                                   *
*                                                                      *
*       NAME: K_AND_L                                                  *
*       DESCRIPTION:                                                   *
*       This module implements the heart of the Kernighan and          *
*       Lin graph partitioning algorithm.  The algorithm               *
*       continues to look for a better partition                       *
*       until none can be found.  At each iteration, "K"               *
*       number of columns are swapped between column groups            *
*       to improve the gain or decrease the |external costs            *
*       -internal cost| value.  When the best partition is             *
*       found, its state is saved.                                     *
*                                                                      *
*       PASSED VARIABLES: size: COLS or COLS/2 partition size          *
*       RETURNS:  NONE                                                 *
*       GLOBAL VARIABLES USED:  carray                                 *
*       GLOBAL VARIABLES CHANGED:  carray                              *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED: calc_D, build_garray, Gmax                     *
*       CALLING MODULES: col_swap                                      *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/



k_and_l(size)
int size;

{

  int selpairs[COLS];
  int kswap;
  int i;
  int temp;
  kswap=1;
  while (kswap>=0)
   {
     calc_D(size);
     build_garray(selpairs,size);
     kswap=Gmax(size);
     if (kswap>=0)
       for (i=0;i<=kswap;i++)
         {
           temp=carray[selpairs[2*i]];
           carray[selpairs[2*i]]=carray[selpairs[2*i+1]];
           carray[selpairs[2*i+1]]=temp;
         }
   }

 return;

}




/*************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME:  CALC_D                                                   *
*       DESCRIPTION:                                                    *
*       Calc_D calculates the Kernighan and Lin "D values"              *
*       for each node(or column) in the graph set(all columns).         *
*       A node's D value is equal to the total external link costs      *
*       minus the total internal link costs for the node.               *
*       The D is calculated for the initial partition created           *
*       by the "partition" module.                                      *
*                                                                       *
*       PASSED VARIABLES: size: COLS or COLS/2 partition size           *
*       RETURNS:  NONE                                                  *
*       GLOBAL VARIABLES USED: costs, BigD, D                           *
*       GLOBAL VARIABLES CHANGED:  BigD, D                              *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED:  NONE                                           *
*       CALLING MODULES: k_and_l                                        *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*      HISTORY:                                                      *
*                                                                    *
*                                                                    *
*********************************************************************/



calc_D(size)
int size;


{

  int E[COLS];
  int II[COLS];
  int link;
  int temp;
  int lo,hi;
  int x,y,yy;
  int i;
  int jx;


  lo=size/2;
  hi=size-1;

  for (i=0;i<=hi;i++)
   {
    E[i]=0;
    II[i]=0;
   }
  for (x=0;x<=(lo-1);x++)
   for (y=lo;y<=hi;y++)
    {
     link=costs[carray[x]][carray[y]];
     E[x]=E[x]+link;
     E[y]=E[y]+link;
    }
  for (x=0;x<=(lo-1);x++)
   for (y=0;y<=(lo-1);y++)
     if (x!=y)
       II[x]=II[x]+costs[carray[x]][carray[y]];
  for (x=lo;x<=hi;x++)
   for (y=lo;y<=hi;y++)
     if (x!=y)
       II[x]=II[x]+costs[carray[x]][carray[y]];
  for (i=0;i<=7;i++)
    BigD[0][i]=(-BIG_NUMBER);
  for (yy=0;yy<=1;yy++)
   {
    y=4*yy;
    for (x=(0+lo*yy);x<=(lo-1+lo*yy);x++)
     {
       D[x]=E[x]-II[x];
       if (D[x]>BigD[0][3+y])
         {
          i=3+y;
          BigD[0][i]=D[x];
          BigD[1][i]=x;
          while ((BigD[0][i]>BigD[0][i-1]) && (i>y))
           {
            for (jx=0;jx<=1;jx++)
             {
               temp=BigD[jx][i];
               BigD[jx][i]=BigD[jx][i-1];
               BigD[jx][i-1]=temp;
             }
             i--;
           }
         }
     }
    }

 return;

}




/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: GMAX                                                       *
*       DESCRIPTION:                                                     *
*       Gmax informs the k_and_l module on how many columns(nodes)       *
*       should be swapped(as listed in the selpairs array) between       *
*       partitions.  The module sums the gains found for each swap       *
*       made by build_garray.  If a positive sum is found at anytime,    *
*       a swap will be made.  The highest positive sum will indicate     *
*       how far to go (K = GXY) down the selpairs array, swapping        *
*       column pairs as indicated.                                       *
*                                                                        *
*       PASSED VARIABLES: size: COLS or COLS/2 partition size            *
*       RETURNS: GXY: the number of swaps to make (K)                    *
*       GLOBAL VARIABLES USED: g                                         *
*       GLOBAL VARIABLES CHANGED: g                                      *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                        *
*       HARDWARE INPUT:                                                       *
*       HARDWARE OUTPUT:                                                      *
*       MODULES CALLED:  NONE                                                 *
*       CALLING MODULES:  k_and_l                                             *
*                                                                             *
*       AUTHOR: PAUL ROSSBACH                                                 *
*       HISTORY:                                                              *
*                                                                             *
*                                                                             *
*****************************************************************************/



Gmax(size)
int size;


{

 int G;
 int GM;
 int GXY;
 int i;


 G=0;
 GM=0;

 for (i=0;i<=(size/2-1);i++)
   {
     G=G+g[i];
     if (G>GM)
      {
       GM=G;
       GXY=i;
      }
   }
 if (GM==0)
   GXY=(-1);

 return(GXY);

}



/*****************************************************************************
*                                                                             *
*                                                                             *
*       DATE: 1 DEC 1985                                                    *
*       VERSION: 1.0                                                        *
*                                                                           *
*       NAME: BUILD_GARRAY                                                  *
*       DESCRIPTION:                                                        *
*       Build g array fills in a selpairs array that lists the nodes        *
*       that are to be swapped (possibly) by the algorithm, and the         *
*       order in which they are to be swapped.  The k_and_l algorithm       *
*       will eventually tag all the nodes to be swapped between the         *
*       two partitions, resulting in a mirror image of the starting         *
*       partitions.  Therefore, selpairs is an array as big as the          *
*       number of nodes in the set(both partitions).  The nodes to          *
*       be swapped next are selected by high_gn and Recalc_D updates        *
*       the D values.  The D values used and placed in selpairs are         *
*       cleared(-BIG_NUMBER) so recalc_D doesn't waste its time.            *
*                                                                           *
*       PASSED VARIABLES: size: COLS or COLS/2 partition size               *
*                     selpairs: pairs of columns selected for swap          *
*       RETURNS: NONE                                                       *
*       GLOBAL VARIABLES USED:  NONE                                        *
*       GLOBAL VARIABLES CHANGED:    NONE                                   *
*       FILES READ:                                                         *
*       FILES WRITTEN:                                                      *
*       HARDWARE INPUT:                                                     *
*       HARDWARE OUTPUT:                                                    *
*       MODULES CALLED: high_gn, Recalc_D                                   *
*       CALLING MODULES: k_and_l                                            *
*                                                                           *
*       AUTHOR: PAUL ROSSBACH                                               *
*       HISTORY:                                                            *
*                                                                           *
*                                                                           *
****************************************************************************/



build_garray(selpairs,size)
int selpairs[COLS];
int size;

{

  int cpair[2];
  int index;
  int i;

   for (index=0;index<=(size/2-1);index++)
     {
       high_gn(index,cpair);
       for (i=0;i<=1;i++)
        {
          selpairs[index*2+i]=cpair[i];
          D[cpair[i]]=(-BIG_NUMBER);
         }
        if (index!=(size/2-1))
          Recalc_D(cpair,size);
      }

 return;

}




/***********************************************************************
*                                                                      *
*                                                                      *
*       DATE: 1 DEC 1985                                               *
*       VERSION: 1.0                                                   *
*                                                                      *
*       NAME:  HIGH_GN                                                 *
*       DESCRIPTION:                                                   *
*       This module finds the two nodes to swap next by calculating    *
*       which swap will yield the highest gain.  To calculate the      *
*       high gain, the module uses the 8 biggest D values(4 from       *
*       each partition) to see which of up to 16 possible combinations *
*       of 2-node swaps gives the highest gain. These two nodes are    *
*       placed in cpair and the gain is placed in the g array          *
*                                                                      *
*       PASSED VARIABLES:  index: number of swap entries in g already  *
*                          cpair: holding area for the 2 next cols     *
*       RETURNS: NONE                                                  *
*       GLOBAL VARIABLES USED:  g, BigD, costs, carray                 *
*       GLOBAL VARIABLES CHANGED:  g                                   *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED:  NONE                                          *
*       CALLING MODULES:  build_garray                                 *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/



high_gn(index,cpair)
```

```
int index;
int cpair[2];

{

 int i,j;
 int temp;

 g[index]=(-BIG_NUMBER);
 for (i=0;i<=3;i++)
  if (BigD[1][i]>=0)
    for (j=0;j<=3;j++)
      if (BigD[1][4+j]>=0)
       {
        temp=BigD[0][i]+BigD[0][4+j]-
             2*costs[carray[BigD[1][i]]][carray[BigD[1][4+j]]];
        if (temp>g[index])
          {
            g[index]=temp;
            cpair[0]=BigD[1][i];
            cpair[1]=BigD[1][4+j];
          }
       }

 return;

 }



/**************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME: RECALC_D                                                    *
*       DESCRIPTION:                                                      *
*       Recalc_D performs the same function as calc_D, but does it        *
*       after the graph partitioning scheme has been changed.  When       *
*       a tentative swap or tagging of 2 nodes(columns) is made           *
*       between the two partitions, the new D values for each node        *
*       (column) can be updated from the old D values.  Recalc_D          *
*       uses this method whereas Calc_D must calculate each node's        *
*       (column's) D value from scratch.  As with Calc_D, the 4           *
*       highest D values for each partition are saved to be used          *
*       in the next attempt to find a better partition by swapping        *
*       2 elements.  The negative values are filled in the Big_D          *
*       array to indicate a non-valid entry when less than 4 nodes        *
*       remain "untagged for swap" in each partition.                     *
*                                                                        *
*       PASSED VARIABLES: size: COLS or COLS/2 partition size            *
*                         cpair: last 2 columns chosen                   *
*       RETURNS: NONE                                                    *
*       GLOBAL VARIABLES USED:  carray, BigD, D, costs                   *
*       GLOBAL VARIABLES CHANGED:   BigD, D                              *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES: build_garray                                    *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/


Recalc_D(cpair,size)
int cpair[2];
int size;

{
  int temp;
  int offset;
  int A,B;
  int i,jx;
  int x,y,yy;
  int lo,hi;


   offset=size/2;
   A=carray[cpair[0]];
   B=carray[cpair[1]];

   for (i=0;i<=7;i++)
     {
      BigD[0][i]=(-BIG_NUMBER);
      BigD[1][i]=(-1);
     }
   for (yy=0;yy<=1;yy++)
     {
      y=4*yy;
      lo=0+offset*yy;
      hi=(offset-1)+offset*yy;
      for (x=lo;x<=hi;x++)
        if (D[x]>(-BIG_NUMBER))
         {
```

```
                    if (y)
                     D[x]=D[x]+2*costs[carray[x]][B]-2*costs[carray[x]][A];
                    else
                     D[x]=D[x]+2*costs[carray[x]][A]-2*costs[carray[x]][B];
                    if (D[x]>BigD[0][3+y])
                      {
                        i=3+y;
                        BigD[0][i]=D[x];
                        BigD[1][i]=x;
                        while ((BigD[0][i]>BigD[0][i-1]) && (i>y))
                         {
                          for (jx=0;jx<=1;jx++)
                           {
                            temp=BigD[jx][i];
                            BigD[jx][i]=BigD[jx][i-1];
                            BigD[jx][i-1]=temp;
                           }
                          i--;
                         }
                      }
                    }
                }

          return;

         }
```

APPENDIX C "DRAINS" CODE

```
/*********************************************************************
*                                                                   *
*                                                                   *
*       DATE: 1 DEC 1985                                            *
*       VERSION: 1.0                                                *
*                                                                   *
*       TITLE: DRAIN PULLER/OPTIMIZER FOR XROM                      *
*       FILENAME: DRAINS.C                                          *
*       COORDINATOR: CPT LINDERMAN                                  *
*       PROJECT: XROM OPTIMIZER                                     *
*       OPERATING SYSTEM: UNIX V 4.2                                *
*       LANGUAGE: C                                                 *
*       USE: included in gen_XROM.c                                 *
*       CONTENTS:                                                   *
*                       drains()                                    *
*                       L_and_K()                                   *
*                       order_in_col()                              *
*                       fill_col_pairs_array()                      *
*                       fill_row_drains_array()                     *
*                       generate_tour()                             *
*                       smallest_yi()                               *
*                       valid_yis()                                 *
*                       max_value()                                 *
*                       best_gain()                                 *
*                       violation()                                 *
*                       path_track()                                *
*                       spec_path_track()                           *
*                       linkjoin()                                  *
*                       new_links()                                 *
*                       fill_invalid()                              *
*                       tl_random()                                 *
*                       clear()                                     *
*                       choose_x1()                                 *
*                       initialize()                                *
*                       swap()                                      *
*                       look_up()                                   *
*                       hamiltonian_path()                          *
*                                                                   *
*                                                                   *
*       FUNCTION: This program minimizes the number of drains       *
*                 in an XROM array(using the Lin & Kernighan        *
*                 Traveling Salesman Problem Algorithm)and          *
*                 passes the output to the layout.c program.        *
*                                                                   *
*********************************************************************/

/*
#include "stdio.h"
```

```
#define COLS                          48
#define ROWS                         144
#define GROUPS                         4
#define DATAWIDTH                     12
#define IN_SIZE          ROWS*GROUPS*8
#define OUT_SIZE       ROWS*GROUPS*DATAWIDTH
#define KR_L_TIMES                     3
#define KR_L                           1
#define FALSE                          0
#define TRUE                           1
#define MAXROWS                      255
#define MAXDATAWIDTH                  15
*/


#define AL_L                           0
#define BL_R                           1
#define RIRJ                           0
#define RILJ                           1
#define LILJ                           2
#define LIRJ                           3

#define BACKTRACK                      0
#define AGAIN                          1
#define SWAP                           2
#define YI_EQ_YSTAR                    3
#define ALL                            0
#define Y_1                            1
#define Y_2                            2
#define NOT_SEEN_YET                   0
#define SEEN                           1

#define BIG_NUMBER             100000000


extern char out_array[];
extern int sub_c_array[];
extern int permut_col[];
extern int sideways[];
extern int row_permutation[];

int no_pull_row[ROWS][ROWS][2];
int non_col_pairs[DATAWIDTH][DATAWIDTH][4];
int low[5][3][4];
int T6_flag;
int VIO_flag;
int G_valid[5][4];
int G[ROWS+1];
struct cell *t[ROWS+1];
int G_best;
int k;
int f_t;
```

```
struct cell {
       struct cell *al_l_link;
       struct cell *bl_r_link;
       int al_l_weight;
       int bl_r_weight;
       int node_number;
       int ycurrent;
       int xcurrent;
       struct cell *ystar;
       int ystar_weight;
       struct cell *tent_a_link;
       struct cell *tent_b_link;
       int t_a_weight;
       int t_b_weight;
       int al_l_is_y;
       int bl_r_is_y;
       int oldx1;
       int oldx2;
       int two_ycurrent;
       int two_xcurrent;
} nodes[ROWS];



/************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME: DRAINS                                                    *
*       DESCRIPTION:                                                    *
*       This module calls all necessary subroutines to perform the      *
*       drain minimization in the XROM array.                           *
*                                                                       *
*       PASSED VARIABLES:  NONE                                         *
*       RETURNS:  NONE                                                  *
*       GLOBAL VARIABLES USED:  f_t                                     *
*       GLOBAL VARIABLES CHANGED: f_t                                   *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: order_in_col,fill_col_pairs_array,L_and_K       *
*                       hamiltonian_path,fill_row_drains_array          *
*       CALLING MODULES: gen_XROM                                       *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
*************************************************************************/



drains()

{
  int i;
  static min_f_t;


  order_in_col();

  for (i=0;i<=3;i++)
   {
     fill_col_pairs_array(i);
     L_and_K(DATAWIDTH,i);
     hamiltonian_path(DATAWIDTH,i);
   }

  fill_row_drains_array();
  min_f_t = BIG_NUMBER;

  for (i=0;i<=3;i++)
   {
     L_and_K(ROWS,i);  /* i not currently used in call but meant */
                       /* for "avoid checkout option" see L&K ref*/
     if (f_t < min_f_t)
      {
       min_f_t = f_t;
       hamiltonian_path(ROWS,i);
      }
   }


  return;


}


/*************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME:  L_AND_K                                                 *
*       DESCRIPTION:                                                   *
*       This module implements the Lin and Kernighan traveling         *
*       salesman promblem (TSP) Algorithm described in Operations      *
*       Research, pp 498-516, vol 21, no 2, 1973.  The purpose         *
*       or the module is to find a row or column arrangement           *
*       that allows the most drain devices to be extracted             *
*       within an XROM.                                                *
*                                                                      *
*       PASSED VARIABLES: size: ROWS/COLS                              *
*                         time: not used - for future improvements     *
*       RETURNS: NONE                                                  *
*       GLOBAL VARIABLES USED: G_best, G, VIO_flag, T6_flag,           *
*       GLOBAL VARIABLES CHANGED:  G_best, G, VIO_flag, T6_flag,       *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED: initialize, generate_tour, t1_random           *
*                       choose_x1, fill_invalid, smallest_yi           *
*                       valid_yis, path_track, max_value               *
*                       violation, best_gain, swap, clear              *
*                                                                      *
*       CALLING MODULES:                                               *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/


L_and_K(size,time)
int size;
int time;

{

 int all_done,x1_done;
 int progress;
 int swapping;
 int no_yis_swap;
 int result;
 int invalid[4];
 int i;


 initialize();
 f_t=generate_tour(size);

 swapping = TRUE;
```

```
while (swapping == TRUE)
{
  G_best = G[0] = 0;
  swapping = FALSE;
  no_yis_swap = FALSE;
  all_done = t1_random(size,0);
  while ( all_done != TRUE)
  {
    x1_done=choose_x1(0);
    while ((G_best == 0) && ( x1_done != TRUE))
    {
      i=1;
      fill_invalid(invalid,t[2]->node_number);
      smallest_yi(i,size,t[2]->node_number,t[2]->ycurrent,invalid);
      if ( valid_yis(size,i) == TRUE)
      {
        while ((G_best == 0) && (maxvalue(size,i) != FALSE))
        {
          if (path_track(1,t[3],t[4]) == FALSE)
            VIO_flag = TRUE;
          else
            VIO_flag = FALSE;
          progress = TRUE;
          while ( progress == TRUE )
          {
            i += 1;
            fill_invalid(invalid,t[2*i]->node_number);
            smallest_yi(i,size,t[2*i]->node_number,t[2*i]->ycurrent,
                                                          invalid);
            if ( valid_yis(size,i) == TRUE)
            {
              result=BACKTRACK;
              while ((result == BACKTRACK) &&
                     (maxvalue(size,i) != FALSE))
              {
                if ( VIO_flag == TRUE || T6_flag == TRUE )
                  result = violation(size,i);
                else
                 {
                  result = best_gain(size,i,0);
                  if ( i == 2 && result == SWAP )
                    result = BACKTRACK;
                 }
                if (result == SWAP || result == YI_EQ_YSTAR)
                {
                  swap();
                  swapping = TRUE;
                  all_done = TRUE;
                  progress = FALSE;
                }
                if (result == BACKTRACK)
                 {
```

```
                T6_flag = FALSE;
                clear(Y_2);
                i = 2;
               }
           }
           if ( result == BACKTRACK )
            {
            clear(Y_1);
            T6_flag = FALSE;
            VIO_flag = FALSE;
            progress = FALSE;
            i = 1;
            }

         } /* end valid yis if */
        else
        {
          if (i == 2)
             {
               clear(Y_1);
               VIO_flag = FALSE;
               progress = FALSE;
               i = 1;
             }
          else
            {
               if ( VIO_flag || T6_flag )
               {
                 clear(Y_1);
                 T6_flag = FALSE;
                 VIO_flag = FALSE;
                 progress = FALSE;
                 i = 1;
               }
               else
                 no_yis_swap = TRUE;

            }

      if ( no_yis_swap == TRUE )
        {
          swap();
          swapping = TRUE;
          all_done = TRUE;
          progress = FALSE;
        }
      }
  } /* end progress while */
}   /* end maxvalue while */

if (G_best == 0)
  {
      clear(ALL);
      x1_done=choose_x1(1);
      }

    }
    else
    {
      clear(ALL);
      x1_done = choose_x1(1);
    }

  } /* end while x1_done */

  if (all_done != TRUE)
    all_done = t1_random(size,1);

} /* end all_done while */

}  /* end swapping while */


}
```

```
/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: ORDER_IN_COL                                               *
*       DESCRIPTION:                                                     *
*       This module looks at each column-byte of the XROM and            *
*       decides what ordering of the bits within the column yields       *
*       the maximum number of zero pairs.  There are 4 possible          *
*       orderings of bits in each column:                                *
*               a.  leave as is = 0                                      *
*               b.  swap the 2 bit line devices = 1                      *
*               c.  swap the 2 bit line devices and the 2 AO             *
*                   line devices =2                                      *
*               d.  swap the 2 AO line devices =3                        *
*       The module determines which ordering gives the most zero         *
*       pairs by finding the minimun non-zero pair count and             *
*       using that ordering.  The module then places that colmun's       *
*       bits in that order and fills the sub_c_array with each           *
*       column's order.                                                  *
*                                                                        *
*       PASSED VARIABLES: NONE                                           *
*       RETURNS: NONE                                                     *
*       GLOBAL VARIABLES USED: out_array, sub_c_array                     *
*       GLOBAL VARIABLES CHANGED:  out_array, sub_c_array                 *
*       FILES READ:                                                       *
*       FILES WRITTEN:                                                    *
*       HARDWARE INPUT:                                                   *
*       HARDWARE OUTPUT:                                                  *
*       MODULES CALLED: NONE                                              *
*       CALLING MODULES: drains                                           *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
**************************************************************************/


order_in_col()

{
  int sub_col[4][4];
  register i,j;
  unsigned calc,plug1,plug2;
  int compare,next,arrange;
  int x,y;


  /************** clear the temporary counting array         *************/

  for (i=0;i<=COLS-1;i++)
   {
     for (x=0;x<=3;x++)
       for (y=0;y<=3;y++)
          sub_col[x][y]=0;


  /************* for each column, count the non-zero pair    *************/
  /************* for each possible internal bit ordering     *************/

     for (j=0;j<=ROWS-1;j++)
      {
        calc=out_array[i+COLS*j];
        calc=(calc & 0377);
        if (calc & 0140)
          sub_col[0][1]++;
        if (calc & 0102)
          sub_col[0][3]++;
        if (calc & 0220)
          sub_col[1][0]++;
        if (calc & 0030)
          sub_col[1][2]++;
        if (calc & 0044)
          sub_col[2][1]++;
        if (calc & 0006)
          sub_col[2][3]++;
        if (calc & 0201)
          sub_col[3][0]++;
        if (calc & 0011)
          sub_col[3][2]++;
      }


/******* find the best internal column order(most 0-pairs) and *******/
/*******      save that arrangement number in sub_c_array      *******/

  arrange=0;
  compare = (sub_col[0][1]+sub_col[1][2]+sub_col[2][3]);
  next = sub_col[0][3]+sub_col[3][2]+sub_col[2][1];
  if (next < compare)
    {
     compare=next;
     arrange=1;
    }
  next = sub_col[2][3]+sub_col[3][0]+sub_col[0][1];
  if (next < compare)
    {
     compare=next;
     arrange=2;
    }
  next = sub_col[2][1]+sub_col[1][0]+sub_col[0][3];
  if (next < compare)
    {
     compare=next;
     arrange=3;
    }
  sub_c_array[i]=arrange;

  /*******  move the bits in the out_array column to the best order ****/

  if (arrange !=0)
   for (j=0;j<=ROWS-1;j++)
     {
      if (arrange == 1 || arrange == 2)
       {
        calc = out_array[i+COLS*j];
        calc = (calc & 0377);
        plug1= (calc & 060);
        plug2= (calc & 03);
        calc = (calc & 0314);
        plug1 >>= 4;
        plug1 = (plug1 & 03);
        plug2 <<= 4;
        plug2 = (plug1 & 060);
```

```
            calc = (calc ^ plug1 ^ plug2);
            out_array[i+COLS*j]=calc;
           }

          if (arrange == 3 || arrange == 2)
           {
            calc = out_array[i+COLS*j];
            calc = (calc & 0377);
            plug1= (calc & 0300);
            plug2= (calc & 014);
            calc = (calc & 063);
            plug1 >>= 4;
            plug1 = (plug1 & 014);
            plug2 <<= 4;
            plug2 = (plug2 & 0300);
            calc = (calc ^ plug1 ^ plug2);
            out_array[i+COLS*j]=calc;
           }
       }

   }  /* end i loop */

return;

}




/*************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME: FILL_COL_PAIRS_ARRAY                                      *
*       DESCRIPTION:                                                    *
*       This module calculates the N squared divided by 2 minus N       *
*       divided by 2 pairings beteen each column's left and right sides.*
*       The correlation value is the number of non-zero pairs between   *
*       any 2 column sides.  These values are upper triangular since the*
*       correlations are symmetric.  The "distance" measures are used in*
*       the L_and_K algorithm.  The lower-triangle of the matrix is     *
*       filled in also for use later.  The results are placed in the    *
*       non_col_pairs array.                                            *
*                                                                       *
*       PASSED VARIABLES: group: the word # of the XROM                 *
*       RETURNS: NONE                                                   *
*       GLOBAL VARIABLES USED: non_col_pairs, out_array                 *
*       GLOBAL VARIABLES CHANGED: non_col_pairs                         *
*       FILES READ:                                                        *
*       FILES WRITTEN:                                                     *
*       HARDWARE INPUT:                                                    *
*       HARDWARE OUTPUT:                                                   *
*       MODULES CALLED: NONE                                               *
*       CALLING MODULES: drains                                            *
*                                                                          *
*       AUTHOR: PAUL ROSSBACH                                              *
*       HISTORY:                                                           *
*                                                                          *
*                                                                          *
***************************************************************************/



fill_col_pairs_array(group)
int group;


{

 register i,j,x;
 unsigned icalc,jcalc;
 int start;


 /**************      clear the non_col_pairs array       *************/

 for (i=0;i<=DATAWIDTH-1;i++)
    for (j=0;j<=DATAWIDTH-1;j++)
      for (x=RIRJ;x<=LIRJ;x++)
        non_col_pairs[i][j][x]=0;


 /*******    fill the non_col_pairs array by counting the   ********/
 /******* non-zero pairs between columns left & right sides ********/

 start = group*DATAWIDTH;

 for (i=0;i<DATAWIDTH;i++)
   for (j=0;j<DATAWIDTH;j++)

     if (i>j)
      {
       for (x=0;x<=ROWS-1;x++)
        {
         icalc = out_array[COLS*x+i+start];
         jcalc = out_array[COLS*x+j+start];
         icalc = (icalc & 0377);
         jcalc = (jcalc & 0377);
         if (icalc & 01)
```

```
          {
           non_col_pairs[i][j][RIRJ]++;
           non_col_pairs[i][j][RILJ]++;
          }
         else
          {
           if (jcalc & 01)
             non_col_pairs[i][j][RIRJ]++;
           if (jcalc & 0200)
             non_col_pairs[i][j][RILJ]++;
          }

         if (icalc & 0200)
          {
           non_col_pairs[i][j][LIRJ]++;
           non_col_pairs[i][j][LILJ]++;
          }
         else
          {
           if (jcalc & 01)
             non_col_pairs[i][j][LIRJ]++;
           if (jcalc & 0200)
             non_col_pairs[i][j][LILJ]++;
          }

      }

   }


/*******  fill in the other half of the upper-triangular   ********/
/*******  distance matrix keeping track of direction       ********/

 for (i=0;i<=DATAWIDTH-1;i++)
   for (j=0;j<=DATAWIDTH-1;j++)
       if (i<j)
        {
          for (x=RIRJ;x<=LILJ;x += 2)
            non_col_pairs[i][j][x]=non_col_pairs[j][i][x];
          non_col_pairs[i][j][LIRJ]=non_col_pairs[j][i][RILJ];
          non_col_pairs[i][j][RILJ]=non_col_pairs[j][i][LIRJ];


        }


return;

}
```

```
/***********************************************************************
*                                                                     *
*                                                                     *
*       DATE: 1 DEC 1985                                              *
*       VERSION: 1.0                                                  *
*                                                                     *
*       NAME: FILL_ROW_DRAINS_ARRAY                                   *
*       DESCRIPTION:                                                  *
*       The module performs the same function as fill_col_pairs_array *
*       for all the rows of the XROM.  Distance correlations          *
*       for the number of drains which can not be removed if 2 rows   *
*       are placed next to each other are calculated for matches      *
*       about bit line drains and about AO_line draines.  The         *
*       results are placee in the no_pull_row_array.  This            *
*       module must consult the permut_col and sideways arrays        *
*       in order to have the proper bit arrangement.                  *
*                                                                     *
*       PASSED VARIABLES: NONE                                        *
*       RETURNS: NONE                                                 *
*       GLOBAL VARIABLES USED: no_pull_row, sideways, out_array       *
*       GLOBAL VARIABLES CHANGED:  no_pull_row, sideways              *
*       FILES READ:                                                   *
*       FILES WRITTEN:                                                *
*       HARDWARE INPUT:                                               *
*       HARDWARE OUTPUT:                                              *
*       MODULES CALLED: NONE                                          *
*       CALLING MODULES: drains                                       *
*                                                                     *
*       AUTHOR: PAUL ROSSBACH                                         *
*       HISTORY:                                                      *
*                                                                     *
*                                                                     *
***********************************************************************/



fill_row_drains_array()

{

  struct drainlist {
         int  al[4*COLS+1];
         int  bl[4*COLS];

         } d_list[ROWS];

  unsigned calc;
  register i,j,x,y;
```

```
 int rowbyte;
 int save;
 int jj;
 int tempA[4];
 int tempB[4];

/*************      clear the no_pull_row array      ************/

 for (i=0;i<=ROWS-1;i++)
   for (j=0;j<=ROWS-1;j++)
     for (x=AL_L;x<=BL_R;x++)
       no_pull_row[i][j][x]=0;



/********* first fill in each rows drain count for bit  **********/
/********* line drains and AO line drains into d_list   **********/
/********* temp holds the AO(A) and bit(B) counts as    **********/
/********* each row byte is counted- 4 drains to a byte **********/
/********* the counts are then concatenated into d_list **********/

 for (i=0;i<=ROWS-1;i++)
  {
   rowbyte=i*COLS;
   save=0;
   for (j=0;j<=COLS-1;j++)
     {
      for (x=0;x<=3;x++)
        {
         tempA[x]=0;
         tempB[x]=0;
        }
      calc = out_array[rowbyte + permut_col[j]];
      calc = (calc & 0377);

/****** if the byte is flipped, must "look" at it different ******/
/****** from the byte that will not be flipped on output    ******/

      if (sideways[j])
        {
         if ((save) || (calc & 01))
            tempA[0]=1;
         if (calc & 06)
            tempA[1]=1;
         if (calc & 0140)
            tempA[3]=1;
         if (calc & 03)
            tempB[0]=1;
         if (calc & 014)
            tempB[1]=1;
         if (calc & 060)
```

```
                tempB[2]=1;
             if (calc & 0300)
                tempB[3]=1;
             if (calc & 0200)
               save = 1;
             else
                save = 0;
           }
         else
            {
             if ((save) || (calc & 0200))
                tempA[0]=1;
             if (calc & 0140)
                tempA[1]=1;
             if (calc & 060)
                tempA[3]=1;
             if (calc & 0300)
                tempB[0]=1;
             if (calc & 060)
                tempB[1]=1;
             if (calc & 014)
                tempB[2]=1;
             if (calc & 03)
                tempB[3]=1;
             if (calc & 01)
               save = 1;
             else
                save = 0;
           }

         if ( calc & 030)
            tempA[2]=1;

/*********** fill in the d_list for AO & bit - lines    ***************/

       jj=4*j;
         for (y=0;y<=3;y++)
            {
             d_list[i].al[jj+y]=tempA[y];
             d_list[i].bl[jj+y]=tempB[y];
            }
          if (j == COLS-1)
             d_list[i].al[jj+y]=save;

   } /* end j loop */


 } /* end i loop */


/******** fill in the distances between rows for the AO & ************/
```

```
/******** bit - lines in the no_pull_row array          *************/

 for (i=0;i<=ROWS-1;i++)
   for (j=0;j<=ROWS-1;j++)
     if (i > j)
      {
       for (x=0;x<=4*COLS-1;x++)
        {
         if ((d_list[i].al[x] == 1) || (d_list[j].al[x] == 1))
           no_pull_row[i][j][0]++;
         if ((d_list[i].bl[x] == 1) || (d_list[j].bl[x] == 1))
           no_pull_row[i][j][1]++;
        }

         if ((d_list[i].al[4*COLS] == 1) || (d_list[j].al[4*COLS] == 1))
           no_pull_row[i][j][0]++;
      }


/******** fill in the lower triangle of the maxtrix also  *************/

 for (i=0;i<=ROWS-1;i++)
   for (j=0;j<=ROWS-1;j++)
     if (i < j)
       for (x=0;x<=1;x++)
         no_pull_row[i][j][x]=no_pull_row[j][i][x];
return;

}



/***********************************************************************
*                                                                      *
*                                                                      *
*       DATE: 1 DEC 1985                                               *
*       VERSION: 1.0                                                   *
*                                                                      *
*       NAME: GENERATE_TOUR                                            *
*       DESCRIPTION:                                                   *
*       This module will generate a random travelling salesman         *
*       tour of the size passed.  Tours made of size = ROWS must       *
*       have links that start and end (picktype) at the same           *
*       type (ie all links are either AL_L to AL_L =LILJ or            *
*       BL_R to BL_R = RIRJ).  Tours made of sizes = COLS can          *
*       have any of four type links (ie LILJ, LIRJ, RIRJ, or RILJ)     *
*       Both tours must have one link of each type emitting from       *
*       each node. The tour starts and ends at node 0.                 *
*                                                                      *
*       PASSED VARIABLES:  size = ROWS/COLS                            *
*       RETURNS:            totalweight = initial cost of generated tour *
*       GLOBAL VARIABLES USED:nodes                                      *
*       GLOBAL VARIABLES CHANGED: nodes                                  *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: linkjoin                                         *
*       CALLING MODULES: L_and_K                                         *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/



generate_tour(size)
int size;

{

 int num_nodes;
 int starttype;
 int picktype;
 int lasttype;
 int done;
 int almost_done;
 int tonode;
 int weight;
 int mask;
 int total_weight;
 long random();
 int check;



 /*********** determine maximum random number  *********************/

 total_weight=0;
 if (size == DATAWIDTH)
   mask = MAXDATAWIDTH;
 else
   mask = MAXROWS;

 srandom(1);

 /********** start at node 0, pick a linktype *********************/
```

```
done = FALSE;
num_nodes = 0;
lasttype = random();
starttype = lasttype = ( lasttype & 01 );

while( done != TRUE )

/**********  pick a random "to" node        *********************/

 {
   tonode = MAXROWS+1;
   while ((tonode > size-1) || (tonode == 0)
                   || (tonode == num_nodes))

/******** try any node not too big,not = 0,not itself ***********/

     {
       tonode = random();
       tonode = (tonode & mask);
     }

   almost_done = FALSE;
   while (((nodes[tonode].al_l_link != NULL) ||
          (nodes[tonode].bl_r_link != NULL)) && (almost_done < 2))

/******* if the node tried is used, try the next node,etc *******/

     {
       if ( tonode == size-1 )
        {
          tonode = 1;
          almost_done++;
        }
       else
          tonode++;
     }


/******* no nodes remain open so put current link to node 0 *****/

   if ( almost_done == 2)
     {
      tonode = 0;
      picktype = ( starttype ^ 01);
     }
   else
    {
      if ( size == DATAWIDTH )
        {
          picktype = random();
          picktype = ( picktype & 01);
        }
```

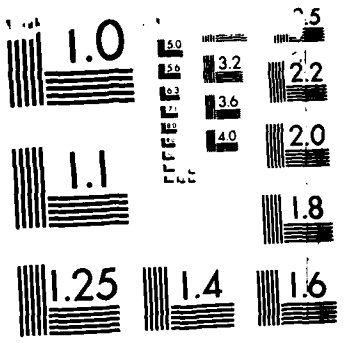
```
            else
               picktype = lasttype;
         }

     /******* join the nodes for the link found using pointers *******/

         if ( picktype == AL_L)
           {
              if (lasttype == AL_L)
                 check = linkjoin(&weight,size,num_nodes,tonode,LILJ);
              else
                 check = linkjoin(&weight,size,num_nodes,tonode,RILJ);
           }

         else
           {
              if (lasttype == AL_L)
                 check = linkjoin(&weight,size,num_nodes,tonode,LIRJ);
              else
                 check = linkjoin(&weight,size,num_nodes,tonode,RIRJ);
           }


         total_weight += weight;


     /********** use the nodes other linktype and continue  **********/

         lasttype = ( picktype ^ 01);
         num_nodes = tonode;
         if (almost_done == 2)
            done = TRUE;

      }

     return(total_weight);

    }




    /************************************************************************
    *                                                                       *
    *                                                                       *
    *       DATE: 1 DEC 1985                                                *
    *       VERSION: 1.0                                                    *
```

```
*                                                                        *
*       NAME: SMALLEST_YI                                                *
*       DESCRIPTION:                                                     *
*       This module will find the smallest five links from any           *
*       "from" node of the "type" specified.  The links can not          *
*       go to any nodes listed in the "invalid"  array.  The             *
*       links are placed into a "low" link array, one for i=i,           *
*       one for i=2, or one for i >= 3.  The low array will              *
*       always be filled.                                                *
*                                                                        *
*       PASSED VARIABLES:     i: current state                           *
*                          size: ROWS/COLS                               *
*                          from: the from node                           *
*                          type: type link needed                        *
*                       invalid: up to 4 illegal "to" nodes              *
*                                                                        *
*       RETURNS: NONE                                                    *
*       GLOBAL VARIABLES USED:low,non_col_pairs,no_pull_row              *
*       GLOBAL VARIABLES CHANGED: low                                    *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES: L_and_K                                         *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/



smallest_yi(i,size,from,type,invalid)
int size;
int from;
int type;
int invalid[4];
int i;

{

 register inc,xx,j;
 int ok;
 int temp;
 int x;
 int ii;
```

```
/*********** fill the low array - for i=1,2, or one for > 3 ***********/

if (i >= 3)
   ii = 3;
else
   ii = i;

for (xx=0;xx<=4;xx++)
    low[xx][0][ii]=BIG_NUMBER;

/*********** find the five smallest legal links from the   *************/
/***********     "from" node of "type" for ROWS problem    *************/

if (size == ROWS)
  {
   for (inc=0;inc<ROWS;inc++)
    {
     ok=TRUE;
     for (j=0;j<=3;j++)
       if (inc == invalid[j])
         ok=FALSE;
     if ((inc != from) && ok)
       {
        if (no_pull_row[from][inc][type] < low[4][0][ii])
          {
           low[4][0][ii]=no_pull_row[from][inc][type];
           low[4][1][ii]=inc;
           low[4][2][ii]=type;
           xx = 4;
           while ((low[xx][0][ii] < low[xx-1][0][ii]) && xx>0)
             {
              for (j=0;j<=2;j++)
                {
                 temp=low[xx][j][ii];
                 low[xx][j][ii]=low[xx-1][j][ii];
                 low[xx-1][j][ii]=temp;
                }
              xx--;
             }
          }
       }
    }
 }

else

/*********** find the five smallest legal links from the   *************/
/***********     "from" node of "type" for COLS problem    *************/

  {
   for (inc=0;inc<DATAWIDTH;inc++)
    {
        ok=TRUE;
        for (j=0;j<=3;j++)
          if (inc == invalid[j])
            ok=FALSE;
        if ((inc != from) && ok)
          for (x=2-type*2;x<=3-type*2;x++)  /* BL_R = 1 :RIRJ,RILJ =0,1*/
          {
           if (non_col_pairs[from][inc][x] < low[4][0][ii])
             {
              low[4][0][ii]=non_col_pairs[from][inc][x];
              low[4][1][ii]=inc;
              low[4][2][ii]=x;
              xx = 4;
              while ((low[xx][0][ii] < low[xx-1][0][ii]) && xx>0)
                {
                  for (j=0;j<=2;j++)
                    {
                     temp=low[xx][j][ii];
                     low[xx][j][ii]=low[xx-1][j][ii];
                     low[xx-1][j][ii]=temp;
                    }
                  xx--;
                }
             }
          }
      }
  }

return;

}



/*********************************************************************
*                                                                   *
*                                                                   *
*       DATE: 1 DEC 1985                                            *
*       VERSION: 1.0                                                *
*                                                                   *
*       NAME: VALID_YIS                                             *
*       DESCRIPTION:                                                *
*       This module looks at the contents of the "low" link array   *
*       and determines which of those 5 links are valid as          *
*       determined by the Lin and Kernighan algorithm.  If a        *
*       proposed link passes all the tests, the ok flag will        *
*       be TRUE and the link is not cleared.  If the link fails     *
*       any test it is cleared from the low  array.  Each Valid yi's *
*       gain values are calculated for one of the tests and saved   *
*       for future use.  If no valid_yi's exist, FALSE is returned      *
*       from the module.                                                *
*                                                                       *
*       PASSED VARIABLES:    i: current state                           *
*                         size: ROWS/COLS                               *
*       RETURNS:          TRUE: at least one valid yi link found        *
*                        FALSE: no valid yi links                       *
*       GLOBAL VARIABLES USED:low, nodes, t, VIO_flag, G_valid, G       *
*       GLOBAL VARIABLES CHANGED: low, G_valid                          *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: path_track,spec_path_track                      *
*       CALLING MODULES: L_and_K                                        *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
************************************************************************/


valid_yis(size,i)
int size;
int i;


{

 struct cell *apt1,*apt2;
 register j,y;
 int ok;
 int test;
 int g_i;
 int ii;


 /********** check the low array - for i=1,2, or one for > 3 ***********/

 if ( i >= 3)
    ii = 3;
 else
    ii = i;

 for (j=0;j<=4;j++)
   if (low[j][0][ii] != -1)   /* for later - don't think this q is nec.*/
     {
      ok=TRUE;
      apt1=(&nodes[low[j][1][ii]]);
```

AD-A164 843   CONTROL CIRCUITRY FOR HIGH SPEED VLSI (VERY LARGE SCALE   4/4
INTEGRATION) WINO.. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI..   P C ROSSBACH
UNCLASSIFIED   DEC 85 AFIT/GE/ENG/85D-35   F/G 9/5   NL

END
FILMED
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
3.2
3.6
4.0

```
/***** determine if the link in low goes to a node not used yet *******/

      if (size == ROWS)
        {
         if (low[j][2][ii] == AL_L)
           if (nodes[low[j][1][ii]].al_l_is_y == 1)
              ok = FALSE;
           else
              apt2=nodes[low[j][1][ii]].al_l_link;
         else
           if (nodes[low[j][1][ii]].bl_r_is_y == 1)
              ok = FALSE;
           else
              apt2=nodes[low[j][1][ii]].bl_r_link;
        }
     else   /** for COLS problem **/
        {
         if (low[j][2][ii] == RIRJ || low[j][2][ii] == LIRJ)
           if (nodes[low[j][1][ii]].bl_r_is_y == 1)
              ok = FALSE;
           else
              apt2=nodes[low[j][1][ii]].bl_r_link;
         else
           if (nodes[low[j][1][ii]].al_l_is_y == 1)
              ok = FALSE;
           else
              apt2=nodes[low[j][1][ii]].al_l_link;
        }

/*********** only a link to t1 can be of t1's "is_y" type  ***********/

     if ((apt1 == t[1]) && (ok == FALSE))
       {
        ok = TRUE;
        apt2 = t[2];
       }

/*********** links can not use t1 as their t[2*i+2] node   ************/

     if ((apt1 == t[1]) && (ok == TRUE))
        ok = FALSE;


/****** if in violation mode, check link to be sure it's legal ********/

     if (apt2 == t[1])
        ok = FALSE;

     if ((i != 1) && (i != 2 || VIO_flag == FALSE) &&
         (i != 3 || T6_flag == FALSE) && ( ok == TRUE ))
       {
```

```
       if ((i == 4 && T6_flag == TRUE) || (i == 3 && VIO_flag == TRUE))
        {
        if (spec_path_track(i,apt1,apt2) == FALSE)
          ok = FALSE;
        }
       else
        {
        if (path_track(i,apt1,apt2) == FALSE)
          ok = FALSE;
        }
      }

/****** the gain must not become negative if the link is placed *******/

    if ( ok == TRUE )
    {

    if (t[2*i]->xcurrent == AL_L)
      g_i = t[2*i]->al_l_weight - low[j][0][ii];
    else
      g_i = t[2*i]->bl_r_weight - low[j][0][ii];


/********** store positive gains in the G_valid array    **************/

    if ((G[i-1] + g_i) > 0)
      G_valid[j][ii] = G[i-1] + g_i;
    else
      ok = FALSE;


    }


/********** if a rule was violated, clear that low entry **************/

    if (ok == FALSE)
       for (y=0;y<=2;y++)
          low[j][y][ii]= -1;



 }


/****** if all low entries are cleared, return FALSE= no valid yis ****/

   test = 0;
   for (j=0;j<=4;j++)
     test += low[j][0][ii];
    if (test == -5)
      return(FALSE);
    else
      return(TRUE);


}




/*************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME: MAX_VALUE                                                 *
*       DESCRIPTION:                                                    *
*       This module finds the link from the valid_yis in low           *
*       that has the maximum | xi+1 - yi | value.  If one is found,     *
*       tentatively assign that link by calling new_link.               *
*                                                                       *
*       PASSED VARIABLES:     i: current state                          *
*                          size: ROWS/COLS                              *
*       RETURNS:           TRUE: max value found from valid yi          *
*                         FALSE: no max value found                     *
*       GLOBAL VARIABLES USED:low, G_valid, G                           *
*       GLOBAL VARIABLES CHANGED: G                                     *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: new_links                                       *
*       CALLING MODULES: L_and_K                                        *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
*************************************************************************/




maxvalue(size,i)
int size;
int i;

{
```

```
int max_yi;
int max_value;
int value;
register j;
int passback;
int ii;


/********** check the low array - for i=1,2, or one for > 3 ***********/

if ( i >= 3)
   ii = 3;
else
   ii = i;

max_yi= -1;
max_value = -BIG_NUMBER;
passback = FALSE;


/******* find the link in low with max |xi+1 - yi| value  *************/

for (j=0;j<=4;j++)
  {
   if (low[j][0][ii] != -1)
    {
      if ((size == ROWS && low[j][2][ii] == BL_R) ||
          (size == DATAWIDTH && (low[j][2][ii] == RIRJ ||
                                 low[j][2][ii] == LIRJ)))

        value=nodes[low[j][1][ii]].bl_r_weight - low[j][0][ii];
      else
        value=nodes[low[j][1][ii]].al_l_weight - low[j][0][ii];

      if (value > max_value)
        {
         max_value=value;
         max_yi=j;
        }
    }
  }


/******** if a max link exists, tentatively fill it in by  ************/
/********                 calling new_links                ************/

if (max_yi != -1)
{

if ((low[max_yi][2][ii] == AL_L && size == ROWS) ||
      (low[max_yi][2][ii] == LILJ && size == DATAWIDTH))

     new_links(LILJ,i,max_yi);

  else
   {
    if ((low[max_yi][2][ii] == BL_R && size == ROWS) ||
        (low[max_yi][2][ii] == RIRJ && size == DATAWIDTH))

     new_links(RIRJ,i,max_yi);

    else
     {
     if (low[max_yi][2][ii] == LIRJ)

         new_links(LIRJ,i,max_yi);

     else

         new_links(RILJ,i,max_yi);
     }
   }


  /************ clear the link used so it can't be used again ***********/

  for (j=0;j<=2;j++)
     low[max_yi][j][ii]= -1;


  /*********** return TRUE since a max was found and set G[i] ***********/

  passback = TRUE;
  G[i] = G_valid[max_yi][ii];

 }

 return(passback);

 }



 /**********************************************************************
 *                                                                     *
 *                                                                     *
 *       DATE: 1 DEC 1985                                              *
 *       VERSION: 1.0                                                  *
 *                                                                     *
 *       NAME: BEST_GAIN                                               *
```

```
*       DESCRIPTION:                                                   *
*       This module determines if the current t_2*i node were         *
*       connected to tl, would a tour with a shorter path (more       *
*       gain) be realized.  If so, the module continues the           *
*       algorithm by connecting the next yi and disconnecting         *
*       the current xi.  If the t_2*i's y_star to tl does not         *
*       improve the tour, SWAP is returned indicating that the        *
*       last best tour should be used.  Best gain also does           *
*       some node "house cleaning" by filling in y_star data at       *
*       the t_2*i, connecting all links permanently, and tagging      *
*       the next t_2*i if not in violation mode.                      *
*                                                                      *
*       PASSED VARIABLES:     i: current state                         *
*                          size: ROWS/COLS                             *
*                          opt_: 0 - normal mode                       *
*                                1 - violation mode                    *
*                                2 - violation - T6 flagged            *
*       RETURNS:           SWAP: t_2*i's ystar gives no gain           *
*                         AGAIN: better tour found, continue           *
*                   YI_EQ_YSTAR: better tour, but yi goes to tl so swap *
*       GLOBAL VARIABLES USED:t, nodes,G_best, G                       *
*       GLOBAL VARIABLES CHANGED: t, nodes,G_best, K                   *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED: NONE                                           *
*       CALLING MODULES: L_and_K                                       *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/


best_gain(size,i,opt_)
int size;
int i;
int opt_;


{

 int last,current,t_one;
 int g_star;
 int type;
 int t_num;
 int b4_last;
 int the_type;
```

```
current=t[2*i]->node_number;
t_one=t[1]->node_number;

/************* fill in ystar data in the nodes structure  *************/

t[2*i]->ystar= (&nodes[t_one]);
if (size == ROWS)
  nodes[current].ystar_weight=
                no_pull_row[current][t_one][nodes[current].ycurrent];
else
 { .
  if (nodes[current].ycurrent == AL_L)
    type = LIRJ;
  else
    type = RIRJ;
  if (nodes[t_one].xcurrent == AL_L)
   {
    if (type == RIRJ)
      type++;
    else
      type--;
   }
  nodes[current].ystar_weight=non_col_pairs[current][t_one][type];
 }



/*********** see if t_2*i's ystar(i) gives a better tour  *************/

if (nodes[current].xcurrent == AL_L)
 g_star= nodes[current].al_l_weight - nodes[current].ystar_weight;
else
 g_star= nodes[current].bl_r_weight - nodes[current].ystar_weight;


/********** if so, save the state and connect the links   *************/

if ((G[i-1] + g_star) > G_best)
  {
   G_best=G[i-1] + g_star;
   k=i;


/************* if i=2, y_l must be connected first        *************/

   if (i == 2 || opt_ != 0)
     {
      if (t[2]->two_ycurrent == AL_L)
        {
         t[2]->al_l_link=t[2]->tent_a_link;
         t[2]->al_l_weight=t[2]->t_a_weight;
```

```
            t[2]->tent_a_link= NULL;
            t[2]->t_a_weight= -1;
            t[2]->al_l_is_y= TRUE;
           }
         else
           {
            t[2]->bl_r_link=t[2]->tent_b_link;
            t[2]->bl_r_weight=t[2]->t_b_weight;
            t[2]->tent_b_link= NULL;
            t[2]->t_b_weight= -1;
            t[2]->bl_r_is_y= TRUE;
           }
         if (t[2]->ycurrent != t[2]->two_ycurrent)
              t[2]->two_ycurrent = t[2]->ycurrent;
         if (t[2]->xcurrent != t[2]->two_xcurrent)
              t[2]->two_xcurrent = t[2]->xcurrent;

        }

/******** connect links for t_2*i(and below if in violation mode ******/

   for (t_num=2*(i-opt_);t_num<=2*i;t_num += 2)
   {
    last=t[t_num-1]->node_number;
    current=t[t_num]->node_number;
    b4_last = t_num-2;


/******** connect the yi link from t_2*i to t_2*i+1 by    *************/
/******** making the tentative links permanent            *************/

    if ( opt_ != 0 || i == 2)
     {
      if (( nodes[current].ycurrent == nodes[current].two_ycurrent ) ||
         ( nodes[current].two_ycurrent == -1 ))
            the_type = nodes[current].ycurrent;
      else
            the_type = nodes[current].two_ycurrent;
     }
    else
       the_type = nodes[current].ycurrent;


    if ( the_type == AL_L)
      {
       if (nodes[current].oldx1 == -1)
         nodes[current].oldx1 = nodes[current].al_l_link->node_number;
       else
         nodes[current].oldx2 = nodes[current].al_l_link->node_number;
       nodes[current].al_l_link=nodes[current].tent_a_link;
       nodes[current].al_l_weight=nodes[current].t_a_weight;
       nodes[current].tent_a_link= NULL;
```

```
          nodes[current].t_a_weight= -1;
          nodes[current].al_l_is_y= TRUE;
         }
       else
         {
          if (nodes[current].oldx1 == -1)
            nodes[current].oldx1 = nodes[current].bl_r_link->node_number;
          else
            nodes[current].oldx2 = nodes[current].bl_r_link->node_number;
          nodes[current].bl_r_link=nodes[current].tent_b_link;
          nodes[current].bl_r_weight=nodes[current].t_b_weight;
          nodes[current].tent_b_link= NULL;
          nodes[current].t_b_weight= -1;
          nodes[current].bl_r_is_y= TRUE;
         }

       if (nodes[current].ycurrent != nodes[current].two_ycurrent)
          nodes[current].two_ycurrent = nodes[current].ycurrent;


   /******** disconnect the xi link from t_2*i-1 to t_2*i and ************/
   /********                connect it to t_2*i-2              ************/

       if ( opt_ != 0 || i == 2)
        {
         if (( nodes[last].xcurrent == nodes[last].two_xcurrent ) ||
            ( nodes[last].two_xcurrent == -1 ))
               the_type = nodes[last].xcurrent;
         else
               the_type = nodes[last].two_xcurrent;
        }
       else
          the_type = nodes[last].xcurrent;


       if ( the_type ==AL_L)
         {
          if (nodes[last].oldx1 == -1)
            nodes[last].oldx1=nodes[last].al_l_link->node_number;
          else
            nodes[last].oldx2=nodes[last].al_l_link->node_number;
          nodes[last].al_l_link= t[b4_last];
          if (t[b4_last]->al_l_link == &nodes[last])
            nodes[last].al_l_weight=t[b4_last]->al_l_weight;
          else
            nodes[last].al_l_weight=t[b4_last]->bl_r_weight;
          nodes[last].al_l_is_y=TRUE;
        }
      else
        {
          if (nodes[last].oldx1 == -1)
            nodes[last].oldx1=nodes[last].bl_r_link->node_number;
            else
              nodes[last].oldx2=nodes[last].bl_r_link->node_number;
            nodes[last].bl_r_link= t[b4_last];
            if (t[b4_last]->al_l_link == &nodes[last])
              nodes[last].bl_r_weight=t[b4_last]->al_l_weight;
            else
              nodes[last].bl_r_weight=t[b4_last]->bl_r_weight;
            nodes[last].bl_r_is_y=TRUE;
          }

           if (nodes[last].xcurrent != nodes[last].two_xcurrent)
                nodes[last].two_xcurrent = nodes[last].xcurrent;

      }

/******* if not in violation mode, tag the next i's t_2*i *************/

      if ( opt_ == 0 )
      {
       if (t[2*i+2]->al_l_link == t[2*i+1])
        {
         t[2*i+2]->xcurrent=AL_L;
         t[2*i+2]->ycurrent=AL_L;
        }
       else
        {
         t[2*i+2]->xcurrent=BL_R;
         t[2*i+2]->ycurrent=BL_R;
        }
      }


      current=t[2*i]->node_number;

/********** if the yi link just placed by best_gain was    *************/
/********** y_star(i) , return that fact                   *************/

      if (((nodes[current].ystar == nodes[current].al_l_link)
            && nodes[current].ycurrent == AL_L) ||
          ((nodes[current].ystar == nodes[current].bl_r_link)
            && nodes[current].ycurrent == BL_R))

                return(YI_EQ_YSTAR);
      else

                return(AGAIN);

    }
```

```
 /******** if the y_star(i) from the current t_2*i did not give ********/
 /******** a gain, don't put in a new tentative link & return SWAP *****/

return(SWAP);


}




/**********************************************************************
*                                                                    *
*                                                                    *
*       DATE: 1 DEC 1985                                             *
*       VERSION: 1.0                                                 *
*                                                                    *
*       NAME: VIOLATION                                              *
*       DESCRIPTION:                                                 *
*       This module handles the cases when the tour's feasibilty     *
*       criteria at i=2 is violated.  If the tour becomes "split"    *
*       at i=2, special constraints have to be placed on the selection *
*       of yi's and backtracking.  This module is called in lew of the *
*       "best gain" module when the VIO_flag is TRUE.                *
*                                                                    *
*       PASSED VARIABLES:     i: current state                       *
*                          size: ROWS/COLS                           *
*       RETURNS:           SWAP: t_2*i's ystar gives no gain         *
*                         AGAIN: better tour found, continue         *
*                   YI_EQ_YSTAR: better tour, but yi goes to tl so swap *
*       GLOBAL VARIABLES USED: t, T6_flag, VIO_flag                  *
*       GLOBAL VARIABLES CHANGED: T6_flag, VIO_flag                  *
*       FILES READ:                                                  *
*       FILES WRITTEN:                                               *
*       HARDWARE INPUT:                                              *
*       HARDWARE OUTPUT:                                             *
*       MODULES CALLED: path_track, best_gain                        *
*       CALLING MODULES: L_and_K                                     *
*                                                                    *
*       AUTHOR: PAUL ROSSBACH                                        *
*       HISTORY:                                                     *
*                                                                    *
*                                                                    *
*********************************************************************/


violation(size,i)
int size;
int i;
```

```
{

 int result;

 switch (i) {

 /********** t3 & t4 caused VIO_flag to be set, check t5 &  ***********/
 /********** t6 to see if they are legal under the VIO_flag ***********/

 case 2:

    if (t[5] == t[1])
      return(BACKTRACK);
    else
     {
      if (path_track(2,t[4],t[5]) == FALSE)
        {
         if (path_track(2,t[5],t[6]) == FALSE)
           return(BACKTRACK);
         else
           T6_flag = TRUE;
        }

     }
  return(AGAIN);

  break;


 /******** if t5 & t6 were placed between t4 & t1 then     ************/
 /******** t7 must be placed between t3 & t2 to close tour ************/


 case 3:

   if (T6_flag)
    {
     if ( t[7] != t[1] )
      {
       if (path_track(3,t[3],t[7]) == FALSE)
          return(BACKTRACK);
       else
          return(AGAIN);
      }
     else
       return(BACKTRACK);
    }
   else

 /********* if t5 & t6 were placed between t3 & t2 then  **************/
 /********* the algorithm is almost back to normal       **************/
```

```
         {
          if ((result=best_gain(size,i,1)) == SWAP)
             return(BACKTRACK);
          else
           {
            VIO_flag = FALSE;
            return(result);
           }
         }


    break;

    /******** here at i=4 if the T6_flag was set           ***************/
    /******** if there's a gain, normal operation, continue ***************/

    default:   /* i=4 */

          if ((result=best_gain(size,i,2)) == SWAP)
             return(BACKTRACK);
          else
           {
             T6_flag = VIO_flag = FALSE;
             return(result);
           }

    break;


   }

   }


   /*************************************************************************
   *                                                                        *
   *                                                                        *
   *       DATE: 1 DEC 1985                                                 *
   *       VERSION: 1.0                                                     *
   *                                                                        *
   *       NAME: PATH_TRACK                                                 *
   *       DESCRIPTION:                                                     *
   *       This module provides the "eyes" to the L_and_K module            *
   *       so it can determine whether or not the link under                *
   *       consideration goes to a point in the tour that is legal.         *
   *       The feasibility criteria is such that if the node pointed to     *
   *       by "node1_ptr" is reached first when following the path          *
   *       from t1, the link under consideration is legal and TRUE          *
   *       is returned.  If the node at "node2_ptr" is reached first, the   *
   *       link is not allowed because it would split the tour into two     *
   *       pieces.                                                          *
```

```
*                                                                      *
*       PASSED VARIABLES:        i: current state                      *
*                        node1_ptr: the pointer to the node that is    *
*                                   closer to t1 in a legal config.    *
*                                   (usually the t_2*i+1 node)         *
*                        node2_ptr: the pointer to the node that is    *
*                                   closer to t1 in an illegal config. *
*                                   (usually the t_2*i+2 node)         *
*       RETURNS:             TRUE: tentative link is legal             *
*                           FALSE: tentative link is illegal           *
*       GLOBAL VARIABLES USED: t, nodes, T6_flag,                      *
*       GLOBAL VARIABLES CHANGED: NONE                                 *
*       FILES READ:                                                    *
*       FILES WRITTEN:                                                 *
*       HARDWARE INPUT:                                                *
*       HARDWARE OUTPUT:                                               *
*       MODULES CALLED: NONE                                           *
*       CALLING MODULES: L_and_K                                       *
*                                                                      *
*       AUTHOR: PAUL ROSSBACH                                          *
*       HISTORY:                                                       *
*                                                                      *
*                                                                      *
***********************************************************************/


path_track(i,node1_ptr,node2_ptr)
int i;
struct cell *node1_ptr,*node2_ptr;

{

  struct cell *track,*next;
  int prev;


/***** if a valid link goes to t1 = legal, adj to t1 = illegal *****/


  if (node1_ptr == t[1])
      return(TRUE);
  if (node2_ptr == t[1])
      return(FALSE);


/***** track along the path from t1 and see what nodes are hit *****/

   track=t[1]->al_l_link;
   if (track == NULL)
     track=t[1]->bl_r_link;
   prev=t[1]->node_number;
   while (track != node1_ptr && track != node2_ptr)
     {

       if (track != t[2*i-1])
         {
          next=track->bl_r_link;
          if (next->node_number != prev && next != NULL)
           {
            prev=track->node_number;
            track=next;
           }
          else
           {
            prev=track->node_number;
            track=track->al_l_link;
           }

          if ((i == 3) && (T6_flag == TRUE) && (track == t[2*i]))
             return(TRUE);

         }
       else

/**** must jump across a tentative link when going from      ****/
/**** t[2*i-1] to t[2*i-2] since the link isn't "there" yet  ****/

        {
         track=t[2*i-2];
         prev=t[2*i-1]->node_number;
         if (T6_flag == TRUE && i == 3)
            prev=t[2*i-3]->node_number;
        }
     }


/**** if arrived at node2 first the tentative tour is illegal ****/


   if (track == node2_ptr)
     return(FALSE);
   else
     return(TRUE);

  }


/**********************************************************************
*                                                                     *
*                                                                     *
*      DATE: 1 DEC 1985                                               *
*       VERSION: 1.0                                                  *
*                                                                     *
*       NAME: SPEC_PATH_TRACK                                         *
*       DESCRIPTION:                                                  *
*       This module performs the same function as "path_track"        *
*       only for special situations.  It is necessary to use a        *
*       special path tracking mechanism when in violation mode        *
*       at i=3 and i=4.  This module capitalizes on the specificity   *
*       of the problem in order to handle it, otherwise it is the     *
*       same as "path_track".                                         *
*                                                                     *
*       PASSED VARIABLES:         i: current state                    *
*                         node1_ptr: the pointer to the node that is  *
*                                    closer to t1 in a legal config.  *
*                         node2_ptr: the pointer to the node that is  *
*                                    closer to t1 in an illegal config. *
*       RETURNS:            TRUE: tentative link is legal             *
*                          FALSE: tentative link is illegal           *
*       GLOBAL VARIABLES USED: t, nodes, T6_flag,                     *
*       GLOBAL VARIABLES CHANGED: NONE                                *
*       FILES READ:                                                   *
*       FILES WRITTEN:                                                *
*       HARDWARE INPUT:                                               *
*       HARDWARE OUTPUT:                                              *
*       MODULES CALLED: NONE                                          *
*       CALLING MODULES: L_and_K                                      *
*                                                                     *
*       AUTHOR: PAUL ROSSBACH                                         *
*       HISTORY:                                                      *
*                                                                     *
*                                                                     *
***********************************************************************/


spec_path_track(i,node1_ptr,node2_ptr)
struct cell *node1_ptr,*node2_ptr;
int i;

{

  struct cell *track,*next;
  int prev;
  int x;
  int t_[8];


/****** if a valid link goes to t1 = legal, adj to t1 = illegal ******/

  if (node1_ptr == t[1])
      return(TRUE);
```

```
    if (node2_ptr == t[1])
        return(FALSE);

  /****** track along the path as in path_track only use many        ****/
  /****** "jumps" over tentative links since many are not permanent ****/


    t_[3]=t_[4]=t_[5]=t_[7]=NOT_SEEN_YET;

     track=t[1]->al_l_link;
     if (track == NULL)
       track=t[1]->bl_r_link;
     prev=t[1]->node_number;

     while (track != node1_ptr && track != node2_ptr)
       {

         for (x=3;x<=7;x++)
           if ((track == t[x]) && (t[x] != NULL))
              t_[x] = SEEN;

         if (((track != t[5] && track != t[6] &&
              (track != t[3] || prev ==t[4]->node_number)
           && (track != t[2] || prev == t[1]->node_number)) && i == 4)  ||

             ((track != t[4] && (track != t[3] || prev ==t[4]->node_number)
           && (track != t[2] || prev == t[1]->node_number)) && i == 3))

           {
            next=track->bl_r_link;
            if (next->node_number != prev && next != NULL)
             {
              prev=track->node_number;
              track=next;
             }
            else
             {
              prev=track->node_number;
              track=track->al_l_link;
             }

         }
         else
           {
             if ((track == t[5]) && (t_[4] == NOT_SEEN_YET))
              {
               track = t[4];
               prev = t[3]->node_number;
              }
             else
              if ((track == t[6]) && (t_[7] == NOT_SEEN_YET))
              {
                track = t[7];
                prev = t[8]->node_number;
               }
               else
                if ((track == t[2]) && (t_[3] == NOT_SEEN_YET))
                {
                 track = t[3];
                 prev = t[4]->node_number;
                }
                else
                 if ((track == t[4]) && (t_[5] == NOT_SEEN_YET))
                  {
                   track = t[5];
                   prev = t[6]->node_number;
                  }
                  else
                   if (track == t[3])
                    {
                     track = t[2];
                     prev = t[1]->node_number;
                    }

           }

      }

      if (track == node2_ptr)
        return(FALSE);
      else
        return(TRUE);



     }




/**************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME: LINKJOIN                                                    *
*       DESCRIPTION:                                                      *
*       Linkjoin is used by the "generate_tour" module when connecting    *
*       nodes with permanent links at the beginning of an L_and_K         *
*       iteration.  The module links the "num_nodes" node to the          *
*       "tonode" node with the "type_link" indicated.  The size           *
*       variable determines which lookup table will be used.              *
*       The weight of the newly placed link is returned.                  *
*                                                                         *
*       PASSED VARIABLES: w_ptr: pointer to weight value location         *
*                          size: ROWS/COLS                                *
*                     num_nodes: the from node                            *
*                        tonode: the to node                              *
*                     type_link: type link to be joined                   *
*       RETURNS:            TRUE: link terminates at open side of a node  *
*                          FALSE: link terminates at a used side of a node *
*       GLOBAL VARIABLES USED: nodes                                      *
*       GLOBAL VARIABLES CHANGED: nodes                                   *
*       FILES READ:                                                       *
*       FILES WRITTEN:                                                    *
*       HARDWARE INPUT:                                                   *
*       HARDWARE OUTPUT:                                                  *
*       MODULES CALLED: lookup                                            *
*       CALLING MODULES: generate_tour                                    *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
**************************************************************************/



linkjoin(w_ptr,size,num_nodes,tonode,type_link)
int *w_ptr;
int size;
int num_nodes;
int tonode;
int type_link;

{

 struct cell **from_side,**to_side;
 int *from_weight,*to_weight;
 int weight;

 /*********** depending on the type of link, load the proper ************/
 /*********** pointer to the correct nodes element           ************/

 if ( type_link == RIRJ || type_link == RILJ)
  {
   from_side = &(nodes[num_nodes].bl_r_link);
   from_weight = &(nodes[num_nodes].bl_r_weight);
  }
 else
  {
   from_side = &(nodes[num_nodes].al_l_link);
```

```
    from_weight = &(nodes[num_nodes].al_l_weight);
   }

  if ( type_link == RIRJ || type_link == LIRJ)
   {
    to_side = &(nodes[tonode].bl_r_link);
    to_weight = &(nodes[tonode].bl_r_weight);
   }
  else
   {
    to_side = &(nodes[tonode].al_l_link);
    to_weight = &(nodes[tonode].al_l_weight);
   }

  /*********** use the pointers to change the nodes links     *************/
  /*********** for the new connection if the node is "open"   *************/

  if (*to_side == NULL)
   {
     weight = look_up(size,num_nodes,tonode,type_link);
     *w_ptr=weight;
     *from_side = (&nodes[tonode]);
     *to_weight = *from_weight = weight;
     *to_side = (&nodes[num_nodes]);
     return(TRUE);
   }
  else
    return(FALSE);

 }




 /***************************************************************************
 *                                                                         *
 *                                                                         *
 *       DATE: 1 DEC 1985                                                  *
 *       VERSION: 1.0                                                      *
 *                                                                         *
 *       NAME: NEW_LINKS                                                   *
 *       DESCRIPTION:                                                      *
 *       New_links sets up all tentative links for the L_and_K             *
 *       algorithm.  The information about the new tentative link          *
 *       is found in the low array.  New_links is passed the index value *
 *       (max_yi) for the low array and the type_link to be installed.   *
 *       New_links also tags the next node as t_2*i+1, and the t_2*i+2   *
 *       node if i=1 or if in the violation mode (best_gain tags t_2*i=2 *
 *       if i>1 and not in the violattion mode).                          *
 *                                                                         *
 *       PASSED VARIABLES: type_link: LILJ,LIRJ,RIRJ,RILJ                 *
 *                                 i: current state                        *
```

```
*                               max_yi: 0-4, low[max_index]             *
*                                                                       *
*       RETURNS: NONE                                                   *
*       GLOBAL VARIABLES USED: t, nodes, VIO_flag                       *
*       GLOBAL VARIABLES CHANGED: t, nodes                              *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: NONE                                            *
*       CALLING MODULES: max_value                                      *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
*************************************************************************/


new_links(type_link,i,max_yi)
int i;
int type_link;
int max_yi;

{
 int ii;



 /********* use the proper low array - for i=1,2, or for > 3 ***********/

 if ( i >= 3)
    ii = 3;
 else
    ii = i;


 /******** tentatively assign the new link from the current  ***********/
 /******** t[2*i] node using the info in low and type_link   ***********/

 if ( type_link == RIRJ || type_link == RILJ)
   {
    t[2*i]->tent_b_link=(&nodes[low[max_yi][1][ii]]);
    t[2*i]->t_b_weight=low[max_yi][0][ii];
    t[2*i]->ycurrent=BL_R;
    t[2*i]->xcurrent=BL_R;
    t[2*i]->bl_r_is_y = TRUE;
    if ( i <= 3)
      {
        if (t[2*i]->two_ycurrent == -1)
            t[2*i]->two_ycurrent = t[2*i]->ycurrent;
        if (t[2*i]->two_xcurrent == -1)
```

```
                t[2*i]->two_xcurrent = t[2*i]->xcurrent;
          }

      }
     else
      {
        t[2*i]->tent_a_link=(&nodes[low[max_yi][1][ii]]);
        t[2*i]->t_a_weight=low[max_yi][0][ii];
        t[2*i]->ycurrent=AL_L;
        t[2*i]->xcurrent=AL_L;
        t[2*i]->al_l_is_y = TRUE;
        if ( i <= 3)
          {
            if (t[2*i]->two_ycurrent == -1)
                t[2*i]->two_ycurrent = t[2*i]->ycurrent;
            if (t[2*i]->two_xcurrent == -1)
                t[2*i]->two_xcurrent = t[2*i]->xcurrent;
          }
      }


    /******** identify the t[2*i+1] and t[2*i+2] nodes reulting ***********/
    /******** from the new tentative yi link                    ***********/
    /******** tentatively assign the t[2*i+1] x & y currents    ***********/

     t[2*i+1]=(&nodes[low[max_yi][1][ii]]);

     if ( type_link == RIRJ || type_link == LIRJ)
      {
        t[2*i+1]->xcurrent=BL_R;
        t[2*i+2]=t[2*i+1]->bl_r_link;
        t[2*i+1]->bl_r_is_y = TRUE;
        if ( i <= 3)
            if (t[2*i+1]->two_xcurrent == -1)
                t[2*i+1]->two_xcurrent = t[2*i+1]->xcurrent;
      }
     else
      {
        t[2*i+1]->xcurrent=AL_L;
        t[2*i+2]=t[2*i+1]->al_l_link;
        t[2*i+1]->al_l_is_y = TRUE;
        if ( i <= 3)
            if (t[2*i+1]->two_xcurrent == -1)
                t[2*i+1]->two_xcurrent = t[2*i+1]->xcurrent;
      }


    /******** if i=1 or the VIO_flag is set, best_gain won't be ***********/
    /****** called so tentatively assign the t[2*i+2] x & y currents ******/

     if ( i == 1 || VIO_flag == TRUE )
      {
        if (t[2*i+2]->al_l_link == t[2*i+1])
         {
          t[2*i+2]->xcurrent=AL_L;
          t[2*i+2]->ycurrent=AL_L;
         }
        else
         {
          t[2*i+2]->xcurrent=BL_R;
          t[2*i+2]->ycurrent=BL_R;
         }
       }


   return;

   }




/************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME: FILL_INVALID                                              *
*       DESCRIPTION:                                                    *
*       This module fills the invalid array with up to 4 nodes that the *
*       current (2*i) node may not choose for a possible 2*i+1 node.    *
*       The nodes may be invalid because they are presently connected   *
*       to the current node or because they were previously connected   *
*       to the node.  This invalid array is used by the smallest_yi     *
*       module each time it is called.                                  *
*                                                                       *
*       PASSED VARIABLES: invalid: ptr to array                         *
*                         nodenum: current(2*i) node                    *
*       RETURNS: NONE                                                   *
*       GLOBAL VARIABLES USED: t,nodes                                  *
*       GLOBAL VARIABLES CHANGED: NONE                                  *
*       FILES READ:                                                     *
*       FILES WRITTEN:                                                  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: NONE                                            *
*       CALLING MODULES: L_and_K                                        *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                          *
***************************************************************************/


fill_invalid(invalid,nodenum)
int invalid[4];
int nodenum;

{

 int i;


 /************** clear invalid                    ****************/

 for (i=0;i<=3;i++)
   invalid[i] = -1;

 /****** enter the node #s of the nodes connected to 2*i ********/

 if (nodes[nodenum].al_l_link != NULL)
   invalid[0] = nodes[nodenum].al_l_link->node_number;
 else
   invalid[0] = t[1]->node_number;

 if (nodes[nodenum].bl_r_link != NULL)
   invalid[1] = nodes[nodenum].bl_r_link->node_number;
 else
   invalid[1] = t[1]->node_number;

 /****** enter node #s of nodes previously connected to 2*i *****/

 if (nodes[nodenum].oldx1 != -1)
   {
    invalid[2] = nodes[nodenum].oldx1;
    if (nodes[nodenum].oldx2 != -1)
       invalid[3] = nodes[nodenum].oldx2;
   }


return;


}




/***************************************************************************
*                                                                          *
*                                                                          *
*       DATE: 1 DEC 1985                                                   *
*       VERSION: 1.0                                                       *
*                                                                          *
*       NAME: T1_RANDOM                                                    *
*       DESCRIPTION:                                                       *
*       This module picks a random starting node, t1, from a list of      *
*       nodes that have not yet been chosen as t1 since the last tour     *
*       change.  If no nodes remain, no node is chosen and TRUE is        *
*       returned.                                                          *
*                                                                          *
*       PASSED VARIABLES: size: ROWS/COLS                                  *
*                         time: 0 = 1st t1 chosen after new tour          *
*                               1 = other t1 tries                         *
*       RETURNS:  TRUE: if all nodes have been tried as t1 w/o gain       *
*                FALSE: if another t1 was found                            *
*       GLOBAL VARIABLES USED:  t1, nodes                                  *
*       GLOBAL VARIABLES CHANGED:  t1, nodes                               *
*       FILES READ:                                                        *
*       FILES WRITTEN:                                                     *
*       HARDWARE INPUT:                                                    *
*       HARDWARE OUTPUT:                                                   *
*       MODULES CALLED: NONE                                               *
*       CALLING MODULES: L_and_K                                           *
*                                                                          *
*       AUTHOR: PAUL ROSSBACH                                              *
*       HISTORY:                                                           *
*                                                                          *
*                                                                          *
***************************************************************************/


tl_random(size,time)
int size;
int time;


{

  static int tl_nodelist[ROWS];
  int possible;
  int mask;
  int done,i;
  int sum;
  long random();


 /*********** determine maximum random number  *********************/

 if (size == DATAWIDTH)
   mask = MAXDATAWIDTH;
```

```
else
  mask = MAXROWS;

/*********** if 1st try, clear the tl nodelist ********************/

if (time == 0)
 {
   for (i=0;i<=size-1;i++)
     tl_nodelist[i] = 0;
 }

/*********** if all nodes have been tried, return TRUE ************/

sum = 0;
for (i=0;i<=size-1;i++)
  sum += tl_nodelist[i];
if ( sum == size )
  return(TRUE);
else

/********** else find a new random tl node    *********************/
 {
  done = FALSE;
  possible = MAXROWS+1;
  while ( possible > size-1 )
    {
     possible = random();
     possible = (possible & mask);
    }

  while (done != TRUE)
   {
    if (tl_nodelist[possible] == 0)
     {
      tl_nodelist[possible] = 1;
      t[1] = (&nodes[possible]);
      done = TRUE;
     }
    else

/******** if random node has been used, use the next(etc) *********/

     {
      if (possible != size-1)
        possible++;
      else
        possible = 0;
     }
   }

  return(FALSE);
 }
```

```
}


/***********************************************************************
*                                                                     *
*                                                                     *
*       DATE: 1 DEC 1985                                              *
*       VERSION: 1.0                                                  *
*                                                                     *
*       NAME: CLEAR                                                   *
*       DESCRIPTION:                                                  *
*       Clear removes all tentative links beyond the point indicated  *
*       by how_much.  Clear is called after a swap is made or after an *
*       attempt to improve the tour has failed, and a new configuration *
*       of the tour is desired.                                       *
*                                                                     *
*       PASSED VARIABLES: how_much - ALL: clear all but t1            *
*                                  - Y_1: clear all back to t2        *
*                                  - Y_2: clear all back to t4        *
*       RETURNS: NONE                                                 *
*       GLOBAL VARIABLES USED: t, nodes                               *
*       GLOBAL VARIABLES CHANGED: t,nodes                             *
*       FILES READ:                                                   *
*       FILES WRITTEN:                                                *
*       HARDWARE INPUT:                                               *
*       HARDWARE OUTPUT:                                              *
*       MODULES CALLED:  NONE                                         *
*       CALLING MODULES: L_and_K                                      *
*                                                                     *
*       AUTHOR: PAUL ROSSBACH                                         *
*       HISTORY:                                                      *
*                                                                     *
*                                                                     *
***********************************************************************/


clear(how_much)
int how_much;

{

 int start,i;
 int reclaim_weight;


 if (how_much == ALL)
   start = 2;
 else
```

```
   if ( how_much == Y_1 )
    start = 3;
   else
    start = 5;

/*********** clear all from start on if changed(ie t[i]!=NULL) *******/

  for (i=start;i<=ROWS-1;i++)
   {
    if ((t[i] != t[2] || start == 2)&&((t[i] != t[3] && t[i] != t[4]) ||
         start != 5) && t[i] != NULL)
     {
       t[i]->xcurrent = -1;
       t[i]->ycurrent = -1;
       t[i]->ystar = NULL;
       t[i]->ystar_weight = -1;
       t[i]->tent_a_link = NULL;
       t[i]->tent_b_link = NULL;
       t[i]->t_a_weight = -1;
       t[i]->t_b_weight = -1;
       t[i]->al_l_is_y = FALSE;
       t[i]->bl_r_is_y = FALSE;
       t[i]->oldx1 = -1;
       t[i]->oldx2 = -1;
       t[i]->two_ycurrent = -1;
       t[i]->two_xcurrent = -1;
       t[i]= NULL;
     }

   else
     {
      if ( t[i] != NULL )
         {
          if (t[i]->two_ycurrent != t[i]->ycurrent)
               t[i]->ycurrent = t[i]->two_ycurrent;
          if (t[i]->two_xcurrent != t[i]->xcurrent)
               t[i]->xcurrent = t[i]->two_xcurrent;
          if (t[i]->ycurrent == AL_L)
             {
              t[i]->tent_b_link = NULL;
              t[i]->t_b_weight = -1;
              if ((t[i] == t[2] && t[i] == t[4]) == FALSE)
                t[i]->bl_r_is_y = FALSE;
             }
          if (t[i]->ycurrent == BL_R)
             {
              t[i]->tent_a_link = NULL;
              t[i]->t_a_weight = -1;
              if ((t[i] == t[2] && t[i] == t[4]) == FALSE)
                t[i]->al_l_is_y = FALSE;
             }
          t[i]= NULL;
```

```
                }
          }

        }


/********* clear Y_1 from node t2 or Y_2 from node t4  **************/


 if (how_much == Y_1)
   i = 2;
 if (how_much == Y_2)
   i = 4;
 if (how_much != ALL)
   {
      if (t[i]->two_ycurrent != t[i]->ycurrent)
          {
           t[i]->ycurrent = t[i]->two_ycurrent;
           t[i]->two_ycurrent = -1;
          }
      if (t[i]->two_xcurrent != t[i]->xcurrent)
          {
           t[i]->xcurrent = t[i]->two_xcurrent;
           t[i]->two_xcurrent = -1;
          }
      if ( t[i]->ycurrent == AL_L)
          {
           t[i]->tent_b_link = NULL;
           t[i]->t_b_weight = -1;
          }
      else
          {
           t[i]->tent_a_link = NULL;
           t[i]->t_a_weight = -1;
          }
   }
 else

/********* clear ALL, relink t1 & t2, then clear both  **************/

  if ( t[1] != NULL )    /* = NULL after swap()  */
   {
     if ( t[1]->al_l_link == NULL)
       {
         t[1]->al_l_is_y = FALSE;
         t[1]->al_l_link = (&nodes[t[1]->oldx1]);
         reclaim_weight = t[1]->al_l_weight;
         }
     else
       {
         t[1]->bl_r_is_y = FALSE;
         t[1]->bl_r_link = (&nodes[t[1]->oldx1]);
```

```
              reclaim_weight = t[1]->bl_r_weight;
              }
          if ( nodes[t[1]->oldx1].al_l_link == NULL)
             {
              nodes[t[1]->oldx1].al_l_link = t[1];
              nodes[t[1]->oldx1].al_l_weight = reclaim_weight;
             }
          else
             {
              nodes[t[1]->oldx1].bl_r_link = t[1];
              nodes[t[1]->oldx1].bl_r_weight = reclaim_weight;
             }
          t[1]->oldx1 = -1;

      }


return;

}




/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: CHOOSE_X1                                                  *
*       DESCRIPTION:                                                     *
*       For any given t1 chosen, choose_x1 will pick a random link as    *
*       the x1 when called with time = 0.  The module will pick the      *
*       remaining link if both have not already been tried with time= 1 *
*       If a link can be assigned, the module assigns it by manipulating*
*       the doubly linked nodes forming the tour.  t1 is split from t2  *
*       and x and y currents, oldx1s, and is_ys are flagged.             *
*                                                                        *
*       PASSED VARIABLES: time : 0 - immediately after t1 chosen         *
*                              : 1 - any other time                      *
*       RETURNS:  FALSE : an x1 assigned                                 *
*                 TRUE  : both x1s used so x1_done                       *
*       GLOBAL VARIABLES USED: t,nodes                                   *
*       GLOBAL VARIABLES CHANGED:   t,nodes                              *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES: L_and_K                                         *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
***************************************************************************/



choose_x1(time)
int time;

{

  static int the_x1;
  static int try;
  long random();


/************* first try - pick random x1         ******************/

  if ( time == 0 )
   {
     the_x1 = random();
     the_x1 = (the_x1 & 01);
     try = 1;
   }
  else

/************* next try - pick the other link     ******************/

   {
     if (try == 1)
      {
        if ( the_x1 == AL_L)
           the_x1 = BL_R;
        else
           the_x1 = AL_L;
        try = 2;
      }
     else

/************* third try - return TRUE = x1_done  ******************/

       return(TRUE);
   }


/********** link assigned - make changes to t and nodes  ***********/

   if ( the_x1 == AL_L)
```

```
          {
            t[2]= t[1]->al_l_link;
            t[1]->al_l_is_y = TRUE;
            t[1]->al_l_link = NULL;
            t[1]->xcurrent = AL_L;
            t[1]->ycurrent = AL_L;
          }
        else
          {
            t[2]= t[1]->bl_r_link;
            t[1]->bl_r_is_y = TRUE;
            t[1]->bl_r_link = NULL;
            t[1]->xcurrent = BL_R;
            t[1]->ycurrent = BL_R;
          }

        if ( t[2]->al_l_link == t[1] )
          {
            t[2]->al_l_is_y = TRUE;
            t[2]->al_l_link = NULL;
            t[2]->xcurrent = AL_L;
            t[2]->ycurrent = AL_L;
          }
        else
          {
            t[2]->bl_r_is_y = TRUE;
            t[2]->bl_r_link = NULL;
            t[2]->xcurrent = BL_R;
            t[2]->ycurrent = BL_R;
          }


      t[2]->oldxl=t[1]->node_number;
      t[1]->oldxl=t[2]->node_number;

    return(FALSE);


    }




/**********************************************************************
*                                                                     *
*                                                                     *
*       DATE: 1 DEC 1985                                              *
*       VERSION: 1.0                                                  *
*                                                                     *
*       NAME: INITIALIZE                                              *
*       DESCRIPTION:                                                  *
*       Initializes all node structures, t pointer arrays, G arrays      *
*       and T6 & VIO _flags.  The initialization either clears or        *
*       places "illegal" values.  Initialize serves to insure all        *
*       unassigned variable values are known at the beginning of the     *
*       algorithm.                                                       *
*                                                                        *
*       PASSED VARIABLES: NONE                                           *
*       RETURNS: NONE                                                    *
*       GLOBAL VARIABLES USED: t,G,nodes,T6_flag,VIO_flag                *
*       GLOBAL VARIABLES CHANGED: t,G,nodes,T6_flag,VIO_flag             *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES: L_and_K                                         *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/


initialize()

{

  int i;


  for (i=0;i<=ROWS-1;i++)
    {
     t[i] = NULL;
     G[i] = 0;


      nodes[i].al_l_link = NULL;
      nodes[i].bl_r_link = NULL;
      nodes[i].al_l_weight = -1;
      nodes[i].bl_r_weight = -1;
      nodes[i].node_number = i;
      nodes[i].ycurrent = -1;
      nodes[i].xcurrent = -1;
      nodes[i].ystar = NULL;
      nodes[i].ystar_weight = -1;
      nodes[i].tent_a_link = NULL;
      nodes[i].tent_b_link = NULL;
      nodes[i].t_a_weight = -1;
      nodes[i].t_b_weight = -1;
      nodes[i].al_l_is_y = FALSE;
```

```
          nodes[i].bl_r_is_y = FALSE;
          nodes[i].oldx1 = -1;
          nodes[i].oldx2 = -1;
          nodes[i].two_ycurrent = -1;
          nodes[i].two_xcurrent = -1;


        }

        T6_flag = FALSE;
        VIO_flag = FALSE;
        t[ROWS] = NULL;
        G[ROWS] = 0;

    return;


    }




    /*************************************************************************
    *                                                                        *
    *                                                                        *
    *       DATE: 1 DEC 1985                                                 *
    *       VERSION: 1.0                                                     *
    *                                                                        *
    *       NAME: SWAP                                                       *
    *       DESCRIPTION:                                                     *
    *       The L_and_K TSP algorithm calls swap when a better tour has      *
    *       found, but it can progress no further.  Since the program        *
    *       changes links as it goes, only the very last valid link needs    *
    *       to be modified.  Then, all tentative information contained in    *
    *       the linked node structures must be cleared.                      *
    *                                                                        *
    *       PASSED VARIABLES: NONE                                           *
    *       RETURNS: NONE                                                    *
    *       GLOBAL VARIABLES USED: f_t,G_best,t                              *
    *       GLOBAL VARIABLES CHANGED: nodes,t                                *
    *       FILES READ:                                                      *
    *       FILES WRITTEN:                                                   *
    *       HARDWARE INPUT:                                                  *
    *       HARDWARE OUTPUT:                                                 *
    *       MODULES CALLED: clear                                            *
    *       CALLING MODULES: L_and_K                                         *
    *                                                                        *
    *       AUTHOR: PAUL ROSSBACH                                            *
    *       HISTORY:                                                         *
    *                                                                        *
```

```
*                                                                           *
****************************************************************************/


swap()

{

 /**************  update the full tour cost     ****************/

  f_t = f_t - G_best;

 /******** move the y(k) link to t1 to close the tour **********/


  if (t[2*k]->al_l_link == t[2*k+1])
   {
    t[2*k]->al_l_link = t[2*k]->ystar;
    t[2*k]->al_l_weight = t[2*k]->ystar_weight;
   }
  else
   {
    t[2*k]->bl_r_link = t[2*k]->ystar;
    t[2*k]->bl_r_weight = t[2*k]->ystar_weight;
   }

 /********  connect t1 back thru y(k) to t[2*k]   **************/


  if (t[1]->al_l_is_y == TRUE)
   {
    t[1]->al_l_link = t[2*k];
    t[1]->al_l_weight = t[2*k]->ystar_weight;
    t[1]->al_l_is_y = FALSE;
   }
  else
   {
    t[1]->bl_r_link = t[2*k];
    t[1]->bl_r_weight = t[2*k]->ystar_weight;
    t[1]->bl_r_is_y = FALSE;
   }

  t[1]->xcurrent = -1;
  t[1]->ycurrent = -1;
  t[1]->oldx1 = -1;
  t[1] = NULL;

  clear(ALL);
```

```
   return;

}



/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: LOOK_UP                                                    *
*       DESCRIPTION:                                                     *
*       Finds and returns the weight of the link between two columns     *
*       or rows from the passed variable information.                    *
*       PASSED VARIABLES:                                                *
*                       size: COLS/ROWS                                  *
*                       from: node                                       *
*                       to:   node                                       *
*                       type: RIRJ                                       *
*                             RILJ                                       *
*                             LILJ                                       *
*                             LIRJ                                       *
*                                                                        *
*       RETURNS: link weight                                             *
*       GLOBAL VARIABLES USED: no_pull_row, non_col_pairs                *
*       GLOBAL VARIABLES CHANGED: NONE                                   *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES: linkjoin                                        *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/



look_up(size,from,to,type)
int size;
int from;
int to;
int type;
```

```
{
 int weight;

 /********  ROWS have BL_R to BL_R or AL_L to AL_L  ************/


  if ( size == ROWS )
  {
    if (type == RIRJ)
      weight = no_pull_row[from][to][BL_R];
    else
      weight = no_pull_row[from][to][AL_L];
  }

 /********  COLS retrieve weight 1 of 4 types   ****************/

  else
      weight = non_col_pairs[from][to][type];


  return(weight);

}






/**************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME: HAMILTONIAN_PATH                                            *
*       DESCRIPTION:                                                      *
*       This module takes the minimum length TSP tour and splits it       *
*       (since the row's or column's array edges need not be joined       *
*       on the ends).  The resulting hamiltonian path is saved in the     *
*       proper array to be used later.  The ordering is temporarily       *
*       saved in the "order" array.  The starting point in the order      *
*       array is determined from either max_start or min_start for        *
*       columns and rows respectively.                                    *
*                                                                         *
*       COLS:  The column tour is split at the maximum link, thereby      *
*       minimizing the costs between the other links.  The resulting      *
*       path for the groups of 12 column-bytes is placed into the        *
*       "permut_col" array.  If the column-byte had to be flipped over   *
*       it is so indicated in the "sideways" array.  The 4 calls to      *
*       this module for columns cover the 4 groups of 12 in the XROM     *
*       from left to right.                                              *
*                                                                        *
*       ROWS:  This module is called when the tour found is the lowest   *
*       seen thus far.  The row tour is only allowed to be split on an   *
*       AL_L(an AO_line drain) link.  This is due to the physical layout*
*       of the XROM.  The rows tour is split on the minimum cost link    *
*       in order to get a two-for-one "drain savings" on the best link   *
*       since this link borders on two rows of drains(top and bottom).   *
*       The resulting path is saved in the "row_permutation" array.      *
*                                                                        *
*                                                                        *
*                                                                        *
*                                                                        *
*       PASSED VARIABLES:                                                *
*                       size: COLS/ROWS                                  *
*                       time: 0,1,2,3 for 4 column groups only           *
*                                                                        *
*       RETURNS: NONE                                                    *
*       GLOBAL VARIABLES USED: nodes                                     *
*       GLOBAL VARIABLES CHANGED: permut_col,sideways,row_permutation    *
*       FILES READ:                                                      *
*       FILES WRITTEN:                                                   *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: NONE                                             *
*       CALLING MODULES: drains                                          *
*                                                                        *
*       AUTHOR: PAUL ROSSBACH                                            *
*       HISTORY:                                                         *
*                                                                        *
*                                                                        *
*************************************************************************/


hamiltonian_path(size,time)
int size;
int time;

{

 int done;
 int max_link;
 int min_aO_link;
 int max_start;
 int min_start;
 int i,j;
int current_type;
int order[ROWS];
int flip[ROWS];
struct cell *follow,*last,*next;


/*************  start at node zero          ******************/

follow = &nodes[0];
last = NULL;
i = 0;
order[i++] = 0;
min_a0_link = BIG_NUMBER;
max_link = -1;

done = FALSE;

/*************  track along the linked tour  ******************/

while ( done != TRUE )
  {
    next = follow->bl_r_link;
    current_type = BL_R;
    flip[i-1] = 0;
    if (next == last)
      {
        next = follow->al_l_link;
        current_type = AL_L;
        flip[i-1] = 1;
      }

/**********  remember the node sequence      ******************/

    if (next != &nodes[0])
       order[i++] = next->node_number;
    else
       i = 1;

/************* if link is AL_L, keep max & min  ****************/

    if ( current_type == AL_L)
      {
        if ( follow->al_l_weight > max_link )
          {
            max_link = follow->al_l_weight;
            max_start = i-1;
          }
        if ( follow->al_l_weight < min_a0_link )
          {
            min_a0_link = follow->al_l_weight;
            min_start = i-1;
          }
```

```
            }
          else

       /************* if the link is BL_R, keep max  ******************/

            {
              if ( follow->bl_r_weight > max_link )
                {
                  max_link = follow->bl_r_weight;
                  max_start = i-1;
                }
            }

       /*************  continue if not finished     ******************/

         if ( next == &nodes[0] )
            done = TRUE;
         else
           {
             last = follow;
             follow = next;
           }

         } /* end while */

       /*************  save the results: ordering   ******************/

       if (size == DATAWIDTH)
        {
          for (j=0;j<=DATAWIDTH-1;j++)
           {
             permut_col[time*DATAWIDTH+j] = order[(max_start +j) % DATAWIDTH]
                                            + time*DATAWIDTH;
             sideways[time*DATAWIDTH+j] = flip[(max_start + j) % DATAWIDTH];
           }
        }
       else
        {
          for (j=0;j<=ROWS-1;j++)
            row_permutation[j] = order[(min_start + j) % ROWS];
        }

      return;

      }
```

APPENDIX D "LAYOUT" CODE

```
/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       TITLE: AUTOMATIC LAYOUT OF XROM IN CAESAR                        *
*       FILENAME: LAYOUT.C                                               *
*       COORDINATOR: CPT LINDERMAN                                       *
*       PROJECT: XROM OPTIMIZER(AUTO LAYOUT)                             *
*       OPERATING SYSTEM: UNIX V 4.2                                     *
*       LANGUAGE: C                                                      *
*       USE: included in gen_XROM.c                                      *
*       CONTENTS:                                                        *
*                       layout()                                         *
*                       pla_pers()                                       *
*                       senseamp_pers()                                  *
*                       xrom_pers()                                      *
*                       main_xrom_place()                                *
*                       edges_xrom_place()                               *
*                       word_sign_pers()                                 *
*                                                                        *
*       FUNCTION: This program uses the results of the DRAINS.C          *
*                 program to create 10 Caesar files that describe        *
*                 the XROM in a layout description.                      *
*                                                                        *
*************************************************************************/

/*
#include "stdio.h"

#define COLS                            48
#define ROWS                           144
#define GROUPS                           4
#define DATAWIDTH                       12
#define FALSE                            0
#define TRUE                             1
#define OUT_SIZE         ROWS*GROUPS*DATAWIDTH
*/

#define LEFT                             0
#define RIGHT                            1
#define MAXCAENUMSIZE                    6

#define TECH                         "scp"
#define CAE_UNITS                        2
#define PLA_PERS_UOLO               "u010"
#define PLA_PERS_UOL1               "u011"
#define PLA_PERS_U1LO               "u110"
```

```
#define PLA_PERS_U1L1                "u1l1"
#define SENSE_AMP_                  "sena_"
#define SIDE                            "s"
#define NOT                             "n"
#define FLIP                            "f"
#define XROM_                       "xrom_"
#define A                               "A"
#define B                               "B"
#define C                               "C"
#define D                               "D"
#define OPT_NO                          "_"
#define FILE_EXT                      ".ca"
#define LEFT_PLA_OUT                 "lpla.ca"
#define RIGHT_PLA_OUT                "rpla.ca"
#define SIGN1_BITS_OUT               "1sig.ca"
#define SIGN2_BITS_OUT               "2sig.ca"
#define XROM1L_OUT                   "Xar1l.ca"
#define XROM1R_OUT                   "Xar1r.ca"
#define XROM2L_OUT                   "Xar2l.ca"
#define XROM2R_OUT                   "Xar2r.ca"
#define SENSE1L_OUT                  "SA1l.ca"
#define SENSE1R_OUT                  "SA1r.ca"
#define SENSE2L_OUT                  "SA2l.ca"
#define SENSE2R_OUT                  "SA2r.ca"
#define XROM_WIDTH                   12*CAE_UNITS
#define XROM_HIEGHT                  12*CAE_UNITS
#define SENSE_WIDTH                  98*CAE_UNITS
#define SENSE_SPACING                96*CAE_UNITS
#define PLA_WIDTH                    13*CAE_UNITS
#define PLA_HIEGHT                   26*CAE_UNITS
#define PLA_SPACING                  25*CAE_UNITS
#define NUM_PLA_ADDRS                    8


extern char out_array[];
extern char word_sign_bit[];
extern int col_sign_bit[];
extern int sub_c_array[];
extern int permut_col[];
extern int sideways[];
extern int row_permutation[];
extern int carray[];

char *plaboxsize;
char *senseboxsize;
char *xromboxsize;


/**************************************************************************
*                                                                         *
*                                                                          *
*       DATE: 1 DEC 1985                                                   *
*       VERSION: 1.0                                                       *
*                                                                          *
*       NAME:  LAYOUT                                                      *
*       MODULE NUMBER:                                                     *
*       DESCRIPTION:                                                       *
*       This module simply calls the four personalization modules         *
*       that create the output caesar files.                               *
*                                                                          *
*       PASSED VARIABLES:  NONE                                            *
*       RETURNS:  NONE                                                     *
*       GLOBAL VARIABLES USED: plaboxsize, senseboxsize, xromboxsize       *
*       GLOBAL VARIABLES CHANGED: plaboxsize, senseboxsize, xromboxsize *
*       FILES READ:                                                        *
*       FILES WRITTEN:                                                     *
*       HARDWARE INPUT:                                                    *
*       HARDWARE OUTPUT:                                                   *
*       MODULES CALLED:  pla_pers, senseamp_pers, xrom_pers,               *
*                        word_sign_pers                                    *
*       CALLING MODULES: gen_XROM                                          *
*                                                                          *
*       AUTHOR: PAUL ROSSBACH                                              *
*       HISTORY:                                                           *
*                                                                          *
*                                                                          *
***************************************************************************/


layout()

{
  plaboxsize = "0 0 26 52";        /* xmin ymin xmax ymax */
  senseboxsize = "0 0 196 200";
  xromboxsize = "0 0 24 24";
   /*  remember caesar units are CAE_UNITS*(lamda_length) */

  pla_pers();
  senseamp_pers();
  xrom_pers();
  word_sign_pers();

  return;

}

/***************************************************************************
*                                                                          *
*                                                                          *
*       DATE: 1 DEC 1985                                                   *
*       VERSION: 1.0                                                       *
*                                                                          *
*       NAME:  PLA_PERS                                                    *
*       MODULE NUMBER:                                                     *
*       DESCRIPTION:                                                       *
*       This module creates the personalization caesar files for           *
*       side of the XROM arrays (left and right).  The proper 1's          *
*       and 0's for the NAND pla are placed in the correct positions       *
*       as determined by placement and drains.                             *
*                                                                          *
*       PASSED VARIABLES:  NONE                                            *
*       RETURNS:  NONE                                                     *
*       GLOBAL VARIABLES USED: row_permutation                             *
*       GLOBAL VARIABLES CHANGED: NONE                                     *
*       FILES READ:                                                        *
*       FILES WRITTEN:  LEFT_PLA_OUT, RIGHT_PLA_OUT                        *
*       HARDWARE INPUT:                                                    *
*       HARDWARE OUTPUT:                                                   *
*       MODULES CALLED:  NONE                                              *
*       CALLING MODULES: layout                                            *
*                                                                          *
*       AUTHOR: PAUL ROSSBACH                                              *
*       HISTORY:                                                           *
*                                                                          *
*                                                                          *
***************************************************************************/



pla_pers()

{

  int i,j,x;
  int calc1,calc2;
  FILE *fp,*fopen();
  char *tech,*cell;
  char *ext;


  ext  = FILE_EXT;
  tech = TECH;
  for (i=LEFT;i<=RIGHT;i++)
   {
     if ( i == LEFT)
       {
        fp = fopen(LEFT_PLA_OUT,"w");
        fprintf(fp,"tech %s\n",tech);
       }
     else
       {
        fprintf(fp,"<<end>>\n");
```

```
          fclose(fp);
          fp = fopen(RIGHT_PLA_OUT,"w");
          fprintf(fp,"tech %s\n",tech);
         }

/********** scan through the row_permutation array 2 rows at a *********/
/********** time and alternating 2 rows per side.  The row     *********/
/********** number in row_permutation equals the programmed    *********/
/**********            address for the NAND pla.               *********/

     for (j=ROWS-4+2*i;j>=2*i;j -= 4)
       {
        calc1 = row_permutation[j];
        calc2 = row_permutation[j+1];
        for (x=0;x<NUM_PLA_ADDRS;x++)
          {
            if (calc1 & 01)
              if (calc2 & 01)
                cell = PLA_PERS_U1L1;
              else
                cell = PLA_PERS_U0L1;
            else
              if (calc2 & 01)
                cell = PLA_PERS_U1L0;
              else
                cell = PLA_PERS_U0L0;
            fprintf(fp,"use %s%s\n",cell,ext);
            fprintf(fp,"transform 1 0 %d 0 1 %d\n",x*PLA_WIDTH,
                                       ((ROWS/4-1)-j/4)*PLA_SPACING);
            fprintf(fp,"box %s\n",plaboxsize);

            calc1 = (calc1 >> 1);
            calc2 = (calc2 >> 1);
          }  /* end x for */
       }     /* end j for */
   }         /* end i for */

  fprintf(fp,"<<end>>\n");
  fclose(fp);

 return;

}



/***************************************************************************
*                                                                         *
*                                                                         *
*       DATE: 1 DEC 1985                                                  *
*       VERSION: 1.0                                                      *
*                                                                         *
*       NAME:  SENSEAMP_PERS                                              *
*       MODULE NUMBER:                                                    *
*       DESCRIPTION:                                                      *
*       This module creates the caesar files for the sense amplifier     *
*       arrays for each XROM array.  The sense amp cells called          *
*       and placed depend on whether the preceding optimization has      *
*       flipped the column over, inverted the column, moved the column,  *
*       and/or changed the column's internal order, and whether the      *
*       XROM array is to the left or right of the center cell.  The      *
*       module also labels each cell with the column's bit position      *
*       number obtained from permut_col since the columns are scrambled.*
*                                                                         *
*       PASSED VARIABLES:  NONE                                           *
*       RETURNS:  NONE                                                    *
*       GLOBAL VARIABLES USED: sideways, col_sign_bit, sub_c_array,      *
*                              permut_col.                                *
*       GLOBAL VARIABLES CHANGED:                                         *
*       FILES READ:                                                       *
*       FILES WRITTEN: SENSE1L_OUT,SENSE1R_OUT,SENSE2R_OUT,SENSE2L_OUT   *
*       HARDWARE INPUT:                                                   *
*       HARDWARE OUTPUT:                                                  *
*       MODULES CALLED:  NONE                                             *
*       CALLING MODULES:  layout                                          *
*                                                                         *
*       AUTHOR: PAUL ROSSBACH                                             *
*       HISTORY:                                                          *
*                                                                         *
*                                                                         *
***************************************************************************/


senseamp_pers()

{

  int i,j,x,y;
  int tag;
  int xpos;
  int col_num;
  char *append1;
  char *append2;
  char *append3;
  char *tech;
  char *ext;
  char *cell;
```

```
    FILE *fp,*fopen();

    cell = SENSE_AMP_;
    tech = TECH;
    ext  = FILE_EXT;

/*********** for 2 sets of left and right XROM arrays **************/

    for (i=1;i<=2;i++)
      for (j=LEFT;j<=RIGHT;j++)
        {
         if (i == 1 && j == LEFT)
           fp = fopen(SENSE1L_OUT,"w");
         else
           {
            fprintf(fp,"<<end>>\n");
            fclose(fp);
            if ( i == 1)
              fp = fopen(SENSE1R_OUT,"w");
            else
              if ( j == LEFT)
                fp = fopen(SENSE2L_OUT,"w");
              else
                fp = fopen(SENSE2R_OUT,"w");
           }
         fprintf(fp,"tech %s\n",tech);


/****** call and position the proper sense amp cell configuation ******/

         for (y=0;y<DATAWIDTH;y++)
           {
            x = (2*(i-1)+j)*DATAWIDTH +y;

            if ( sideways[x] )
              append1 = SIDE;
            else
              append1 = OPT_NO;

            if ( col_sign_bit[permut_col[x]])
              append2 = NOT;
            else
              append2 = OPT_NO;

            if ( j == RIGHT )
              append3 = FLIP;
            else
              append3 = OPT_NO;

            tag = sub_c_array[permut_col[x]];

            fprintf(fp,"use %s%s%s%s%d%s\n",cell,append1,append2,
                                                            append3,tag,ext);

                fprintf(fp,"transform 1 0 %d 0 1 0\n",y*SENSE_SPACING);

                fprintf(fp,"box %s\n",senseboxsize);

               }  /* end y for */

             fprintf(fp,"<< labels >>\n");
             for (y=0;y<DATAWIDTH;y++)
               {
                xpos = (y+.5)*SENSE_WIDTH;
                x = (2*(i-1)+j)*DATAWIDTH +y;
                col_num = carray[permut_col[x]];
                fprintf(fp,"label %d %d %d %d %d 1\n",col_num,xpos,0,xpos,0);
               }

            }  /* end j for */

        fprintf(fp,"<<end>>\n");
        fclose(fp);

      return;

     }



/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: XROM_PERS                                                  *
*       MODULE NUMBER:                                                   *
*       DESCRIPTION:                                                     *
*       This module opens and closes the output files, initiates the     *
*       written output, and calls the two main routines that place       *
*       the XROM personalization cells into the four arrays.             *
*                                                                        *
*       PASSED VARIABLES:  NONE                                          *
*       RETURNS:  NONE                                                   *
*       GLOBAL VARIABLES USED: NONE                                      *
*       GLOBAL VARIABLES CHANGED:  NONE                                  *
*       FILES READ:                                                      *
*       FILES WRITTEN: XROM1L_OUT, XROM1R_OUT, XROM2L_OUT, XROM2R_OUT    *
*       HARDWARE INPUT:                                                  *
*       HARDWARE OUTPUT:                                                 *
*       MODULES CALLED: main_xrom_place, edges_xrom_place                *
*       CALLING MODULES: layout                                            *
*                                                                          *
*       AUTHOR: PAUL ROSSBACH                                              *
*       HISTORY:                                                           *
*                                                                          *
*                                                                          *
***************************************************************************/


xrom_pers()

{

 int group;
 char *tech;
 FILE *fp,*fopen();


  tech = TECH;

  for (group=0;group<=3;group++)
    {
     if (group == 0)
       fp = fopen(XROM1L_OUT,"w");
     else
       {
        fprintf(fp,"<<end>>\n");
        fclose(fp);
        if (group == 1)
          fp = fopen(XROM1R_OUT,"w");
        else
          if (group == 2)
            fp = fopen(XROM2L_OUT,"w");
          else
            fp = fopen(XROM2R_OUT,"w");
       }
     fprintf(fp,"tech %s\n",tech);

     main_xrom_place(group,fp);
     edges_xrom_place(group,fp);

    }

  fprintf(fp,"<<end>>\n");
  fclose(fp);

 return;

}
```

```
/************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME:  MAIN_XROM_PLACE                                          *
*       MODULE NUMBER:                                                  *
*       DESCRIPTION:                                                    *
*       This module places the XROM cells for most of the array groups  *
*       from the out_array bit pattern and row_permutation matrix.      *
*                                                                       *
*       PASSED VARIABLES: group: 0-3, datawidth wide groups             *
*                         fp: pointer to output file                    *
*       RETURNS:  NONE                                                  *
*       GLOBAL VARIABLES USED: row_permutation, out_array               *
*       GLOBAL VARIABLES CHANGED:                                       *
*       FILES READ:                                                     *
*       FILES WRITTEN: XROM1L_OUT, XROM1R_OUT, XROM2L_OUT, XROM2R_OUT   *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: NONE                                            *
*       CALLING MODULES: xrom_pers                                      *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
************************************************************************/


main_xrom_place(group,fp)
int group; /* 0,1,2,3 */
FILE *fp;

{

 int bit1[9],bit2[9],bit3[9];
 int calc1,calc2,calc3;
 int g,h,i,j,y,x,yy,xx;
 int ur,ul,lr,ll;
 char *a,*b,*c,*d;
 char *cell;
 char *ext;
 int x_dim,y_dim;


  cell = XROM_;
  ext  = FILE_EXT;
```

```
/********** for each XROM row in the group by 2's *************/

for (x=ROWS-1;x>=3;x -= 2)
   {
    bit1[8] = 0;
    bit2[8] = 0;
    bit3[8] = 0;
    g=row_permutation[x-2]*COLS + group*DATAWIDTH;
    h=row_permutation[x-1]*COLS + group*DATAWIDTH;
    i=row_permutation[x]*COLS + group*DATAWIDTH;
    for (j=DATAWIDTH-1;j>=0;j--)
      {


/********* look at three rows at a time   **************/

       calc1 = out_array[g+j];
       calc2 = out_array[h+j];
       calc3 = out_array[i+j];

       for (y=0;y<=7;y++)
         {
          bit1[7-y] = (calc1 & 01);
          bit2[7-y] = (calc2 & 01);
          bit3[7-y] = (calc3 & 01);
          calc1 = (calc1 >> 1);
          calc2 = (calc2 >> 1);
          calc3 = (calc3 >> 1);
         }


/********* calculate type XROM cell for 4       *************/
/********* bits of each bit line(xx=0) byte & 4 ************/
/********* bits of each A0  line(xx=1) byte     ************/

       for (xx=0;xx<=1;xx++)
         for (yy=4;yy>=1;yy--)
           {

            if (xx)
              {
               ur = bit3[2*yy];    /** ur = upper right of drain **/
               ul = bit3[2*yy-1];  /** ul = upper left  of drain **/
               lr = bit2[2*yy];    /** lr = lower right of drain **/
               ll = bit2[2*yy-1];  /** ll = lower left  of drain **/
              }
            else
              {
               ur = bit2[2*yy-1];
               ul = bit2[2*yy-2];
               lr = bit1[2*yy-1];
              ll = bitl[2*yy-2];
            }

          if (ul)
            a = A;
          else
            a = OPT_NO;
          if (ur)
            b = B;
          else
            b = OPT_NO;
          if (ll)
            c = C;
          else
            c = OPT_NO;
          if (lr)
            d = D;
          else
            d = OPT_NO;

          x_dim = XROM_WIDTH*(yy + 4*j + .5*xx);
          y_dim = XROM_HIEGHT*(((ROWS-1)-x)/2 + .5*xx);

          fprintf(fp,"use %s%s%s%s%s%s\n",cell,a,b,c,d,ext);
          fprintf(fp,"transform 1 0 %d 0 1 %d\n",x_dim,y_dim);
          fprintf(fp,"box %s\n",xromboxsize);

        }  /* end yy for */

     bitl[8] = bitl[0];
     bit2[8] = bit2[0];
     bit3[8] = bit3[0];

    } /* end j for */

  }  /* end x for */

 return;

}



/*************************************************************************
*                                                                        *
*                                                                        *
*       DATE: 1 DEC 1985                                                 *
*       VERSION: 1.0                                                     *
*                                                                        *
*       NAME: EDGES_XROM_PLACE                                          *
*       MODULE NUMBER:                                                  *
*       DESCRIPTION:                                                    *
*       This module places the XROM cells for the boundaries of the    *
*       XROM groups that were initially placed by main_xrom_place.     *
*                                                                       *
*       PASSED VARIABLES: group: 0-3, datawidth wide groups            *
*                         fp: pointer to output file                   *
*       RETURNS:  NONE                                                  *
*       GLOBAL VARIABLES USED: row_permutation, out_array              *
*       GLOBAL VARIABLES CHANGED:                                       *
*       FILES READ:                                                    .*
*       FILES WRITTEN: XROM1L_OUT, XROM1R_OUT, XROM2L_OUT, XROM2R_OUT  *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: NONE                                            *
*       CALLING MODULES: xrom_pers                                      *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                           *
*       HISTORY:                                                        *
*                                                                       *
*                                                                       *
************************************************************************/


edges_xrom_place(group,fp)
int group; /* 0,1,2,3 */
FILE *fp;

{


 int bit0[9],bit1[9],bitR_1[9];
 int calc0,calc1,calcR_1;
 int j,y,x,yy,xx;
 int ur,ul,lr,ll;
 int bit[ROWS+2];
 char *a,*b,*c,*d;
 char *cell;
 char *ext;
 int x_dim,y_dim;


  cell = XROM_;
  ext  = FILE_EXT;


 /********* fill in two upper and one lower rows of XROM ********/


  bit0[8] = 0;
```

```
bit1[8] = 0;
bitR_1[8] = 0;

for (j=DATAWIDTH-1;j>=0;j--)
  {

   calc0 = out_array[row_permutation[0]*COLS + group*DATAWIDTH + j];
   calc1 = out_array[row_permutation[1]*COLS + group*DATAWIDTH + j];
   calcR_1=out_array[row_permutation[ROWS-1]*COLS+group*DATAWIDTH+j];

   for (y=0;y<=7;y++)
     {
      bit0[7-y] = (calc0 & 01);
      bit1[7-y] = (calc1 & 01);
      bitR_1[7-y] = (calcR_1 & 01);
      calc0 = (calc0 >> 1);
      calc1 = (calc1 >> 1);
      calcR_1 = (calcR_1 >> 1);
     }

   for (xx=0;xx<=2;xx++)    /** 0=row 0; 1=row1; 2=row ROW-1 **/
     for (yy=4;yy>=1;yy--)
       {

           if (xx == 0)     /** top AO_drains row **/
             {
              ur = 0;
              ul = 0;
              lr = bit0[2*yy];
              ll = bit0[2*yy-1];
             }

           if (xx == 1)     /** top bit_drains row **/
             {
              ur = bit0[2*yy-1];
              ul = bit0[2*yy-2];
              lr = bit1[2*yy-1];
              ll = bit1[2*yy-2];
             }

           if (xx == 2)     /** bottom AO_drains row **/
             {
              ur = bitR_1[2*yy];
              ul = bitR_1[2*yy-1];
              lr = 0;
              ll = 0;
             }

           if (ul)
             a = A;
           else
             a = OPT_NO;
```

```
                if (ur)
                  b = B;
                else
                  b = OPT_NO;
                if (ll)
                  c = C;
                else
                  c = OPT_NO;
                if (lr)
                  d = D;
                else
                  d = OPT_NO;

                if (xx == 1)
                  {
                   x_dim = XROM_WIDTH*(yy + 4*j);
                   y_dim = XROM_HIEGHT*(ROWS-2)/2;
                  }
                else
                  {
                   x_dim = XROM_WIDTH*(yy + 4*j + .5);
                   y_dim = XROM_HIEGHT*(((ROWS-2)-(xx/2)*ROWS)/2 + .5);
                  }

                fprintf(fp,"use %s%s%s%s%s%s\n",cell,a,b,c,d,ext);
                fprintf(fp,"transform 1 0 %d 0 1 %d\n",x_dim,y_dim);
                fprintf(fp,"box %s\n",xromboxsize);


          }  /* end yy for */

        bit0[8] = bit0[0];
        bit1[8] = bit1[0];
        bitR_1[8] = bitR_1[0];

     } /* end j for */


    /********* fill in the left most column of A0-drains ***********/

     for (x=0;x<=ROWS-1;x++)
        bit[ROWS-x] = (out_array[group*DATAWIDTH +
                       COLS*row_permutation[x]] & 0200);
     bit[0] = 0;
     bit[ROWS+1] = 0;

     for (x=0;x<=ROWS;x += 2)
       {

        if (bit[x])
          d = D;
        else
        d = OPT_NO;

      if (bit[x+1])
        b = B;
      else
        b = OPT_NO;

      a = OPT_NO;
      c = OPT_NO;

      x_dim = XROM_WIDTH*(.5);
      y_dim = XROM_HIEGHT*(-.5 + x/2);

      fprintf(fp,"use %s%s%s%s%s%s\n",cell,a,b,c,d,ext);
      fprintf(fp,"transform 1 0 %d 0 1 %d\n",x_dim,y_dim);
      fprintf(fp,"box %s\n",xromboxsize);

   }  /* end x for */


 return;

}


/************************************************************************
*                                                                       *
*                                                                       *
*       DATE: 1 DEC 1985                                                *
*       VERSION: 1.0                                                    *
*                                                                       *
*       NAME:  WORD_SIGN_PERS                                           *
*       MODULE NUMBER:                                                  *
*       DESCRIPTION:                                                    *
*       This module places the word sign bits for each large portion    *
*       of the XROM(XROM 1 & XROM 2) into two caesar files.  Each       *
*       word sign bit column contains 4 sign bits per row (2 for        *
*       each AO/AON in each group).                                     *
*                                                                       *
*       PASSED VARIABLES:  NONE                                         *
*       RETURNS:  NONE                                                  *
*       GLOBAL VARIABLES USED: word_sign_bit,row_permutation            *
*       GLOBAL VARIABLES CHANGED: NONE                                  *
*       FILES READ:                                                     *
*       FILES WRITTEN: SIGN1_BITS_OUT,SIGN2_BITS_OUT                    *
*       HARDWARE INPUT:                                                 *
*       HARDWARE OUTPUT:                                                *
*       MODULES CALLED: NONE                                            *
*       CALLING MODULES: layout                                         *
*                                                                       *
*       AUTHOR: PAUL ROSSBACH                                                  *
*       HISTORY:                                                               *
*                                                                              *
*                                                                              *
*******************************************************************************/


word_sign_pers()

{

  int bit[ROWS+2][6];
  int byte;
  int i,j,y,x;
  int offset;
  int ur,ul,lr,ll;
  char *a,*b,*c,*d;
  FILE *fp, *fopen();
  char *tech;
  char *cell;
  char *ext;
  int x_dim,y_dim;


  ext  = FILE_EXT;
  cell = XROM_;
  tech = TECH;


/*********** for both word sign bit column caesar files ************/

  for (j=0;j<=1;j++)
    {
     if (j == 0)
       {
        fp = fopen(SIGN1_BITS_OUT,"w");
        fprintf(fp,"tech %s\n",tech);
       }
     else
       {
        fprintf(fp,"<<end>>\n");
        fclose(fp);
        fp = fopen(SIGN2_BITS_OUT,"w");
        fprintf(fp,"tech %s\n",tech);
       }
     offset = 4 * j;


/*********** fill the bit array with the current column ************/

     for (x=0;x<=ROWS-1;x++)
       {
          byte = word_sign_bit[row_permutation[x]];

          bit[ROWS-x][0] = 0;
          bit[ROWS-x][1] = byte & ( 0200 >> offset );
          bit[ROWS-x][2] = byte & ( 0100 >> offset );
          bit[ROWS-x][3] = byte & ( 0020 >> offset );
          bit[ROWS-x][4] = byte & ( 0040 >> offset );
          bit[ROWS-x][5] = 0;
         }

      for (i=0;i<=3;i++)
        {
         bit[0][i] = 0;
         bit[ROWS+1][i] = 0;
        }


/******* place the AO/AON drains for the word sign bit column ******/

      for (x=0;x<=ROWS;x += 2)
        for (y=0;y<=2;y++)
          {
           ur = bit[x+1][2*y+1];
           ul = bit[x+1][2*y];
           lr = bit[x][2*y+1];
           ll = bit[x][2*y];


           if (ul)
             a = A;
           else
             a = OPT_NO;
           if (ur)
             b = B;
           else
             b = OPT_NO;
           if (ll)
             c = C;
           else
             c = OPT_NO;
           if (lr)
             d = D;
           else
             d = OPT_NO;

           fprintf(fp,"use %s%s%s%s%s%s\n",cell,a,b,c,d,ext);

           fprintf(fp,"transform 1 0 %d 0 1 %d\n",XROM_WIDTH*y,
                                          XROM_HIEGHT*((ROWS-x)/2));

           fprintf(fp,"box %s\n",xromboxsize);
```

```
                }

/******* place the bitline drains for the word sign bit column ******/

     for (x=1;x<=ROWS-1;x += 2)
       for (y=0;y<=1;y++)
         {
          ur = bit[x+1][2*y+2];
          ul = bit[x+1][2*y+1];
          lr = bit[x][2*y+2];
          ll = bit[x][2*y+1];

          if (ul)
            a = A;
          else
            a = OPT_NO;
          if (ur)
            b = B;
          else
            b = OPT_NO;
          if (ll)
            c = C;
          else
            c = OPT_NO;
          if (lr)
            d = D;
          else
            d = OPT_NO;

          x_dim = XROM_WIDTH*(y + .5);
          y_dim = XROM_HIEGHT*((ROWS-(x-1))/2 - .5);

          fprintf(fp,"use %s%s%s%s%s%s\n",cell,a,b,c,d,ext);
          fprintf(fp,"transform 1 0 %d 0 1 %d\n",x_dim,y_dim);
          fprintf(fp,"box %s\n",xromboxsize);

         }

    }

  fprintf(fp,"<<end>>\n");
  fclose(fp);

 return;

}
```

Appendix E

SPICE Model Parameters

```
MODEL CMOSP PMOS LEVEL=2.00000 LD=0.512860U TOX=500.000E-10
- NSUB=2.971614E+14 VTO=0.844293 KP=1/048805E-05 GAMMA =
   0.723071
- PHI=0.600000 UO=100.000 UEXP=0.145531 UCIT=18543.6
- DELTA=2.19030 V M A X=100000. XJ=2.583588E-02
- NFS=1.615644E+12 NEFF=1.001000E-02 NSS=0.00000E=00
   TPG=1.00000
- RSH= 95 CGS0=4E-10 CGDO=4E-10 CJ=2E-4 MJ=0.5 CJSW=9E-10
   MJSW=0.33

MODEL CMOSN NMOS LEVEL=2.00000 LD=0.245423U TOX=500.000E-10
-NSUB=1.000000E+16 VTO=0.932797 KP=2.696667E-05 GAMMA=1.28047
-PHI=0.600000 UO=381.905 UEXP=1.0010000E-03 UCRIT=999000.
-DELTA=1.47242 VMAX=55346.3 XJ=0.145596U LAMDA=2.491255E-02
-NFS=3.727796E+11 NEFF=1.001000E-02 NSS=0.0000E+00 TPG=1.0000
-RSH=25 CGSO=5.2E-10 CGDO=5.2E-10 CJ=3.2E-4 MJ=0.5 CJSW=9E-10
   MJSW=0.33
```

Appendix F

Typical Parasitic Capacitance

Layer-Layer	Thickness	Capacitance
Tox	500-600 Å	5.6-6.7 E-4 pF/u**2
poly-substrate	7000-8500 Å	.39-.48 E-4 pF/u**2
metal-substrate	1.4-1.7 um	.20-.24 E-4 pF/u**2
metal-diff	9000-9500 Å	.35-.37 E-4 pF/u**2
metal2-substrate	2.5-2.9 um	.12-.16 E-4 pF/u**2
metal2-metal	1.1-1.3 um	.26-.31 E-4 pF/u**2
metal2-poly1	700-900 Å	3.9-4.9 E-4 pF/u**2
N+ Junction Area	---	1.6-5.0 E-4 pF/u**2
N+ Junction Side Wall	---	2.0-3.3 E-4 pF/u**2
P+ Junction Area	---	2.8-5.0 E-4 pF/u**2
P+ Junction Side Wall	---	1.6-5.4 E-4 pF/u**2

Resistances and Current Limits

N+ Diff. sheet resistance:	<= 40 ohms/square
P+ Diff. sheet resistance:	<= 100 ohms/square
Poly1 sheet resistance:	<= 30 ohms/square
Poly2 sheet resistance:	<= 40 ohms/square
Metal1 current limit:	0.6 mA/micron
Metal2 current limit:	1.0 mA/micron

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ADA 164 043

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/85D-35	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering; 6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State and ZIP Code)* Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433-6583	7b. ADDRESS *(City, State and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR; 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State and ZIP Code)* Bolling AFB, Washington DC 20332	10. SOURCE OF FUNDING NOS.
11. TITLE *(Include Security Classification)* See Box 19	

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.

12. PERSONAL AUTHOR(S) Paul C. Rossbach, CPT, US Army

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Yr., Mo., Day)*	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1985 December	352

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.
09	02	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Fast Fourier Transforms, Signal Processing, Integrated Circuits, Read Only Memories, Computer Aided Design

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Title: CONTROL CIRCUITRY FOR HIGH SPEED VLSI
WINOGRAD FOURIER TRANSFORM PROCESSORS

Thesis Chairman: Richard W. Linderman, Captain, USAF
Assistant Professor of Electrical Engineering

Approved for public release: IAW AFR 190-1.
[illegible] 16 JAN 86
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓ XX] SAME AS RPT ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard W. Linderman	22b. TELEPHONE NUMBER *(Include Area Code)* 513-255-6913; 22c. OFFICE SYMBOL AFIT/ENG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

The calculation of the Discrete Fourier Transform (DFT) has long been a significant bottleneck in many Digital Signal Processing applications. With the arrival of Very Large Scale Integration ~~(VLSI)~~ and new DFT algorithms, system architectures that significantly reduce the DFT bottleneck are possible. This study addresses the design, simulation, implementation, and testing of the control circuitry for a high speed, VLSI Winograd Fourier Transform (WFT) processor. Three WFT processors are combined into a pipelined architecture that is capable of computing a 4080-point DFT on complex input data approximately every 120 microseconds when operating with 70 MHz clock signals. The chip control architecture features a special Programmable Logic Array (PLA) to control the on-chip arithmetic circuitry, and a dense, 54K ROM to generate data addresses for the external RAM. The PLA controller was fabricated in 3 micron CMOS and functioned properly for clock rates of over 60 MHz. The address generator ROM was designed and submitted for fabrication in 3 micron CMOS, and SPICE simulations predict an access time of 60 nanoseconds.

Software that automatically generates a ROM layout description from a data file was developed to ensure the correctness of the final design. Software was also developed to optimize the ROM by attempting to minimize the number of transistors required to represent the information. The software further optimizes the ROM by removing any unnecessary drain/source areas. The transistor minimization procedure is based on a graph partitioning heuristic, and the drain removal procedure is based on an algorithm that near-optimally solves the Traveling Salesman Problem. The ROM optimization produces large gains in power, yield, and in some cases speed.